Departament d'Enginyeria de Serveis
i Sistemes d'Informació
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Development of the conceptual schema of a bowling game system by applying TDCM

Research Report

Albert Tort Pugibet
atort@essi.upc.edu

January 2011

Contents

1. Introduction

This document reports a case study application of Test-Driven Conceptual Modeling (TDCM) in the development of the conceptual schema of a bowling game system.

In this section, we briefly review the TDCM method and we present the case study.

1.1. On Test-Driven Conceptual Modeling

Conceptual schemas of information systems can be tested [6]. This essentially means that there is a testing language, in which the conceptual modeler writes programs that test the conceptual schema, and a testing environment in which test programs are executed. In this report we will use the testing language called CSTL (Conceptual Schema Testing Language) [5,6] in order to specify conceptual test cases. A CSTL processor prototype will be used to execute the test cases.

TDCM is an iterative method aimed to drive the elicitation and the definition of the conceptual schema of an information system. TDCM uses test cases to drive the conceptual modeling activity. In TDCM, conceptual schemas are incrementally defined and continuously validated.

A test case written in a conceptual schema testing language is an executable concrete story of a user-system interaction. A test case also specifies user expectations formalized as test assertions. The verdict of a test case is *Pass* if the conceptual schema includes the general knowledge to meet these user expectations.

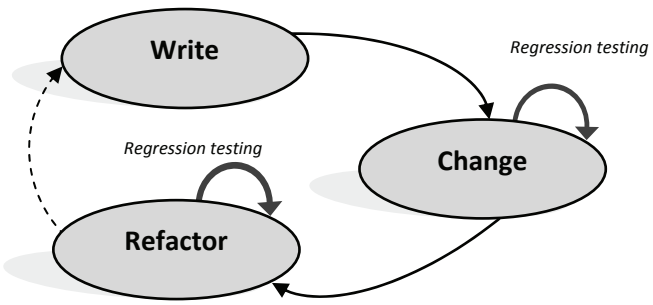


Fig. 1. The TDCM cycle

In TDCM, a conceptual schema is defined incrementally in short iterations. An iteration starts by adding a *new test case* to the passing test set of the previous iteration (*previous test set*). The objective of each iteration is to change the schema so that it includes the knowledge to correctly execute the *new test case*. The *previous test set* including the *new test case* is the *current test set* of the iteration. A TDCM iteration can only finish when the overall verdict of the *current test set* is *Pass*.

Figure 2 shows the TDCM cycle and the order of its tasks. A TDCM iteration is a particular instantiation of the TDCM cycle. The TDCM cycle consists of three kinds of tasks: (1) Write a test case; (2) change the schema; and (3) refactor the schema. We describe these tasks in the following.

1.1.1. Write a Test Case

The first task of the TDCM cycle consists in setting up a new test case whose verdict will be *Pass* once the conceptual schema includes the knowledge to be added in the current iteration.

4.1.2. Change the Schema

Changing the schema to make the verdict of the *new test case Pass* is the focused objective to be achieved by the conceptual modeler in this task. The testing environment provides information about the failure or the error. The next change to be done in the schema is fixing this error or failure.

While changing the schema, test cases of the *previous test set* can be automatically executed as regression tests.

4.1.3. Refactor the Schema

Refactoring is aimed to improve the quality of the conceptual schema without changing the knowledge specified in it. When a conceptual schema is defined in an iterative way, as proposed in TDCM, refactoring may be applied to improve the quality of the schema. TDCM encourages the conceptual modeler to refactor the schema and requesting the execution of the *current test set* after doing that.

If the verdict of the *current test set* becomes *Fail* or *Error*, then we realize that the knowledge of the schema has not been preserved by the refactoring process. The failure/error information provided by the testing environment helps to identify the invalid refactoring changes. If the verdict of the *current test set* is *Pass* and no more refactoring is felt needed to be applied, then we can start a new iteration.

1.2. Case study: A bowling game system

Bowling is a sport in which players attempt to score points by rolling a bowling ball along a flat surface in order to knock down as many pins as possible.

The bowling game system is a popular case study used to demonstrate eXtreme Programming (XP) practices in action. Robert C. Martin popularized this case study in the *Agile Software Develoment* book [3]. A programming episode of this case study is also available online [2]. Martin reviews the rules of bowling as follows:

> The game is played in ten frames. At the beginning of each frame, all ten pins are set up. The player then gets two tries to knock them all down.
>
> If the player knocks all the pins down on the first try, it is called a “strike”, and the frame ends.
>
> If the player fails to knock down all the pins with his first ball, but succeeds with the second ball, it is called a “spare”.
>
> After the second ball of the frame, the frame ends even If there are still pins standing.
>
> A strike frame is cored by adding ten, plus the number of pins knocked down by the next two balls, to the score of the previous frame.
>
> A spare frame is scored by adding ten, plus the number of pins knocked down by the next ball, to the score of the previous frame.
>
> If a strike is thrown in the tenth frame, then the player may throw two more balls to complete the score of the strike.
>
> Likewise, if a spare is thrown in the tenth frame, the player may throw one more ball to complete the score of the spare.

We use the bowling game case study in order to illustrate the application of Test-Driven Conceptual Modeling (TDCM) in the development of the UML/OCL conceptual schema of this system. You may review the main concepts and notation used to define the conceptual schema under development in [6].

2. Testing strategy

The application of TDCM should be supported by following a previously defined testing strategy. Defining a strategy comprises the design of a representative set of test cases which are the input of the TDCM application. The testing strategy should also include criteria to determine the source of the test cases and its order of processing when applying TDCM.

2.1. Strategy overview

The objective of this case study is the development of the conceptual schema of a bowling game system. We have defined eleven user stories (Section 2.3). These user stories are based on the most representative scenarios of the system use cases and the domain rules (Section 2.2). Stories are representative and concrete cases that the stakeholders expect to be able to perform in the information system which is being developed. Each test case processed by using the TDCM method corresponds to one of these stories.

By applying TDCM, once the design test cases succeed, then we can assert that the resultant conceptual schema has the required knowledge according to the user stories.

The stories have been sorted by its expected complexity. This criterion pretends to go with the incremental property of TDCM. For example, when the first story succeeds, then the schema makes feasible games with regular throws. After that, we process stories to add the knowledge about spares and strikes in several common or limit situations.

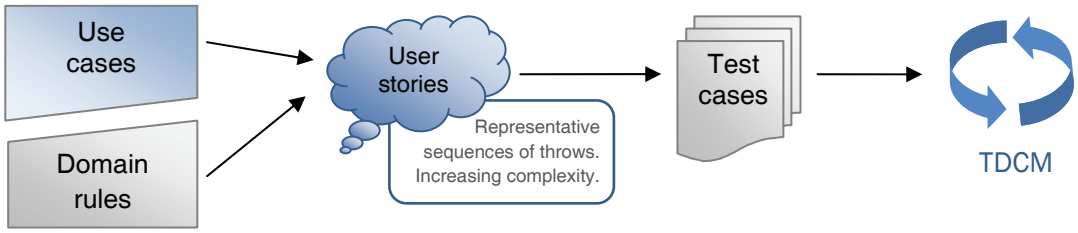


Fig. 2. Testing strategy

2.2. Use cases and domain rules

In the following, we specify the use cases and the domain rules of the bowling game case study.

Use cases

Use case: Start a new game
Identifier: NG.
Preconditions: None.
Trigger: The player wants to start a new bowling game.
Main Success Scenario:
1. The system initializes a new bowling game
   [→NewGame]

Use case: Throw a ball
Identifier: TB.
Preconditions: The game is not finished.
Trigger: The player wants to throw a ball.
Main Success Scenario:
1. The player indicates the number of knocked pins and the game.
2. The system registers the throw.
   [→ RegisterThrow]
3. The system computes the score information.

Domain rules

DR1. A game consists of 10 frames.
DR2. In each frame the player has two throws to knock down 10 pins.
DR3. The score for the frame is the total number of pins knocked down.
DR4. A spare is when the player knocks down all 10 pins in two throws.
DR5. The bonus for a spare frame is the number of pins knocked down by the next throw.
DR6. A strike is when the player knocks down all 10 pins on his first throw.
DR7. The bonus for a strike frame is the number of pins knocked down by the next two throws.
DR8. In the last frame, a player who rolls a spare or strike is allowed to throw the extra balls to complete the frame.

2.3. User stories

In the following, we specify the user stories that will be processed when applying TDCM. Each user story describes a representative or limit case of a sequence of bowling throws.

The design of these stories takes into account the following criteria, which are based on the informal description of the bowling game system that may be found in Section 1.2.:

- Each story represents a level of complexity of the game. The level of complexity comprises two properties: the type of the throws and the frames in which theses throws occur.
    - S1 is a sequence of regular throws (no spares and strikes occur).
    - S2, S3 and S4 add spares complexity in incomplete/complete games and with non-consecutive/consecutive spares.
    - S5, S6 and S7 add strikes complexity (independently from spares) in incomplete/complete games and with non-consecutive/consecutive spares.
    - S8 mixes spares and strikes in a complete game.
    - S9 and S10 are complete games with the particular characteristic that there is a strike or a spare in the last frame.
    - S11 is an extension of S1 in order to assert that no throws are allowed when a game is finished.
- In each throw, a number of pins between 0 and 10 are knocked down. We consider two equivalence classes:
    - Knocking down 0 pins is a limit case of a throw.
    - Knocking down 10 pins is a limit case of a throw.
    - Knocking down 1-9 pins is a regular case of a throw.

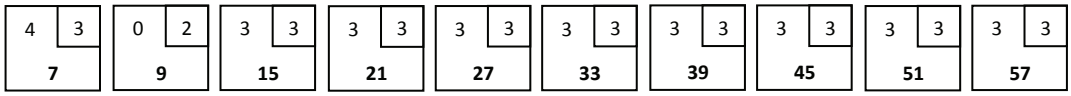
  Each story combines limit throws and regular throws.

According to the incremental nature of TDCM, the processing order of the stories is based on the complexity of the represented game.

The application of Test-Driven Development in the bowling game programming episode reported in [3], no testing strategy is defined prior to TDD application. In this programming episode, stories are processed as they are elicited, by interviewing a stakeholder.

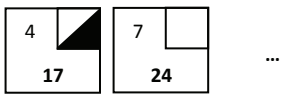
Stories are defined textually as a sequence of steps. Moreover, score cards [1,2] for each sequence of throws are also used in order to graphically represent each story.

S1: A complete game without spares and strikes



- A new game G1 is started. G1 consists of 10 frames.
  Testing objectives: NG, DR1
- A throw knocks down 4 pins (representative case of a first regular throw). The score at frame #1 is 4 and the score of the game is 4.
  Testing objectives: TB, DR2, DR3
- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 7 and the score of the game is 7.
  Testing objectives: TB, DR2, DR3
- The game is not finished.
  Testing objectives: DR1
- A throw knocks down 0 pins (limit case of a first regular throw). The score at frame #1 is 7. The score at frame #2 is 7 and the score of the game is 7.
- **Testing objectives:** TB, DR2, DR3
- A throw knocks down 2 pins (representative case of a second regular throw). The score at frame #1 is 7. The score at frame #2 is 9 and the score of the game is 9.
- **Testing objectives:** TB, DR2, DR3
- 16 regular throws are registered. 3 frames (representative case of a regular throw) are knocked at each throw.
  Testing objectives: TB, DR1, DR2, DR3
- The game is finished.
  Testing objectives: DR1

S2: An incomplete game with a spare in the first frame



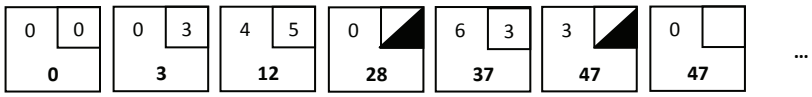
...

- A new game G2 is started.
- A throw knocks down 4 pins (representative case of a first regular throw). The score at frame #1 is 4 and the score of the game is 4.
  Testing objectives: TB, DR2, DR3
- A throw knocks down 6 pins (representative case of a spare). This is a spare. The score at frame #1 is 10 (the spare bonus cannot be added until the next throw score is known) and the score of the game is 10.
  Testing objectives: TB, DR4

- A throw knocks down 7 pins (representative case of a regular throw). The score at frame #1 becomes 17 (the spare bonus is added). The score at frame #2 is 24. The score of the game is 24.
  Testing objectives: TB, DR5

- The game is not finished.
  Testing objectives: DR1

S3: An incomplete game with regular and non-terminal spare frames



- A new game G3 is started.

- A throw knocks down 0 pins (limit case of a first regular throw). The score at frame #1 is 0 and the score of the game is 0.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 0 pins (limit case of a second regular throw). The score at frame #1 is 0 and the score of the game is 0.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 0 pins (limit case of a first regular throw). The score at frame #1 is 0. The score at frame #2 is 0. The score of the game is 0.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 0. The score at frame #2 is 3. The score of the game is 3.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 4 pins (representative case of a first regular throw). The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 7. The score of the game is 7.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 5 pins (representative case of a second regular throw). The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 12. The score of the game is 12.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 0 pins (limit case of a first regular throw). The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 12. The score at frame #4 is 12. The score of the game is 12.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 10 pins (limit case of a spare). This is a spare. The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 12. The score at frame #4 is 22. The score of the game is 22 (the spare bonus cannot be added until the next throw score is known) and the score of the game is 22.
  Testing objectives: TB, DR4

- A throw knocks down 6 pins (representative case of a first regular throw). The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 12. The score at frame #4 is 28. The score at frame #5 is 34.The score of the game is 34.

Testing objectives: TB, DR5

- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 12. The score at frame #4 is 28. The score at frame #5 is 37.The score of the game is 37.
  Testing objectives: TB, DR2, DR3

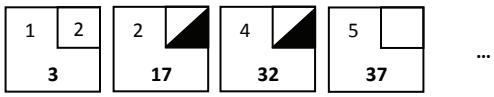
- A throw knocks down 3 pins (representative case of a first regular throw). The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 12. The score at frame #4 is 28. The score at frame #5 is 37. The score at frame #6 is 40. The score of the game is 40.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 7 pins (representative case of a spare). This is a spare. The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 12. The score at frame #4 is 28. The score at frame #5 is 37. The score at frame #6 is 47 (the spare bonus cannot be added until the next throw score is known). The score of the game is 47.
  Testing objectives: TB, DR4

- A throw knocks down 0 pins (limit case of a first regular throw). The score at frame #1 is 0. The score at frame #2 is 3. The score at frame #3 is 12. The score at frame #4 is 28. The score at frame #5 is 37. The score at frame #6 is 47. The score at frame #7 is 47.The score of the game is 47.
  Testing objectives: TB, DR5

- The game is not finished.
  Testing objectives: DR1

S4: An incomplete game with two consecutive and non-terminal spares



- A new game G4 is started.

- A throw knocks down 1 pin (representative case of a first regular throw). The score at frame #1 is 1 and the score of the game is 1.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 2 pins (representative case of a second regular throw). The score at frame #1 is 3 and the score of the game is 3.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 2 pins (representative case of a first regular throw). The score at frame #1 is 3. The score at frame #2 is 5. The score of the game is 5.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 8 pins (representative case of a spare). This is a spare. The score at frame #1 is 3. The score at frame #2 is 13 (the spare bonus cannot be added until the next throw score is known). The score of the game is 13.
  Testing objectives: TB, DR4

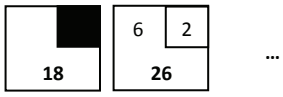
- A throw knocks down 4 pins (representative case of a first regular throw). The score at frame #1 is 3. The score at frame #2 is 17. The score at frame #3 is 21. The score of the game is 21.
  Testing objectives: TB, DR5

- A throw knocks down 6 pins (representative case of a spare). The score at frame #1 is 3. The score at frame #2 is 17. The score at frame #3 is 27 (the spare bonus cannot be added until the next throw score is known). The score of the game is 27.
  Testing objectives: TB, DR4

- A throw knocks down 5 pins (representative case of a first regular throw). The score at frame #1 is 3. The score at frame #2 is 17. The score at frame #3 is 32. The score at frame #4 is 37. The score of the game is 37.
  Testing objectives: TB, DR5

- The game is not finished.
  Testing objectives: DR1

S5: An incomplete game with a strike in the first frame



- A new game G5 is started.

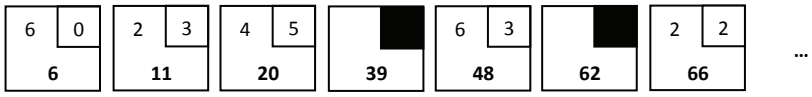
- A throw knocks down 10 pins (spare throw). This is a strike. The score at frame #1 is 10 (the strike bonus cannot be added until the score of the next throws is known). The score of the game is 10.
  Testing objectives: TB, DR6

- A throw knocks down 6 pins (representative case of a first regular throw). The score at frame #1 becomes 16 (the strike bonus is added). The score at frame #2 is 22. The score of the game is 22.
  Testing objectives: TB, DR7

- A throw knocks down 2 pins (representative case of a second regular throw). The score at frame #1 becomes 18 (the strike bonus is added). The score at frame #2 is 26. The score of the game is 26.
  Testing objectives: TB, DR7

- The game is not finished.
  Testing objectives: DR1

S6: An incomplete game with regular and non-terminal strike frames



- A new game G6 is started.

- A throw knocks down 6 pins (representative case of a first regular throw). The score at frame #1 is 6 and the score of the game is 6.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 0 pins (limit case of a second regular throw). The score at frame #1 is 6 and the score of the game is 6.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 2 pins (representative case of a first regular throw). The score at frame #1 is 6. The score at frame #2 is 8. The score of the game is 8.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 6. The score at frame #2 is 11. The score of the game is 11.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 4 pins (representative case of a first regular throw). The score at frame #1 is 6. The score at frame #2 is 11. The score at frame #3 is 15. The score of the game is 15.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 5 pins (representative case of a second regular throw). The score at frame #1 is 6. The score at frame #2 is 11. The score at frame #3 is 20. The score of the game is 20.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 10 pins (strike case). This is a strike. The score at frame #1 is 6. The score at frame #2 is 11. The score at frame #3 is 20. The score at frame #4 is 30 (the strike bonus cannot be added until the score of the next throws is known). The score of the game is 30.
  Testing objectives: TB, DR6

- A throw knocks down 6 pins (representative case of a first regular throw). The score at frame #1 is 6. The score at frame #2 is 11. The score at frame #3 is 20. The score at frame #4 becomes 36 (the strike bonus is added). The score at frame #5 is 42. The score of the game is 42.
  Testing objectives: TB, DR7

- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 6. The score at frame #2 is 11. The score at frame #3 is 20. The score at frame #4 becomes 39 (the strike bonus is added). The score at frame #5 becomes 48. The score of the game is 48.
  Testing objectives: TB, DR7

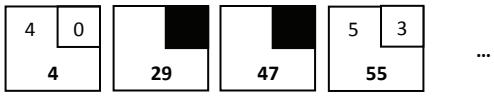
- A throw knocks down 10 pins (strike case). The score at frame #1 is 6. The score at frame #2 is 11. The score at frame #3 is 20. The score at frame #4 is 39. The score at frame #5 is 48. The score at frame #6 is 58 (the strike bonus cannot be added until the score of the next throws is known). The score of the game is 58.
  Testing objectives: TB, DR6

- A throw knocks down 2 pins (representative case of a first regular throw). The score at frame #1 is 6. The score at frame #2 is 11. The score at frame #3 is 20. The score at frame #4 is 39. The score at frame #5 is 48. The score at frame #6 becomes 60 (the strike bonus is added). The score at frame #7 is 62. The score of the game is 62.
  Testing objectives: TB, DR7

- A throw knocks down 2 pins (representative case of a second regular throw). The score at frame #1 is 6. The score at frame #2 is 11. The score at frame #3 is 20. The score at frame #4 is 39. The score at frame #5 is 48. The score at frame #6 becomes 62 (the strike bonus is added). The score at frame #7 becomes 66. The score of the game is 66.
  Testing objectives: TB, DR7

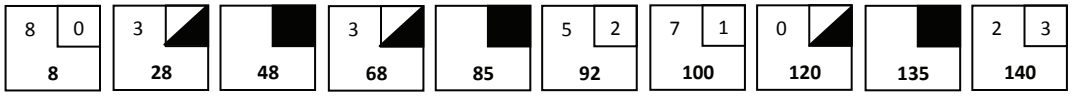
- The game is not finished.
  Testing objectives: DR1

S7: An incomplete game with two consecutive and non-terminal strikes



- A new game G7 is started.
- A throw knocks down 4 pins (representative case of a first regular throw). The score at frame #1 is 4 and the score of the game is 4.
  Testing objectives: TB, DR2, DR3
- A throw knocks down 0 pins (limit case of a second regular throw). The score at frame #1 is 4 and the score of the game is 4.
  Testing objectives: TB, DR2, DR3
- A throw knocks down 10 pins (strike case). This is a strike. The score at frame #1 is 4. The score at frame #2 is 14 (the strike bonus cannot be added until the score of the next throws is known). The score of the game is 14.
  Testing objectives: TB, DR6
- A throw knocks down 10 pins (strike case). This is a strike. The score at frame #1 is 4. The score at frame #2 becomes 24 (the strike bonus is added). The score at frame #3 is 34 (the strike bonus cannot be added until the score of the next throws is known). The score of the game is 34.
  Testing objectives: TB, DR6, DR7
- A throw knocks down 5 pins (representative case of a first regular throw). The score at frame #1 is 4. The score at frame #2 becomes 29 (the strike bonus is added). The score at frame #3 becomes 44 (the strike bonus is added). The score at frame #4 becomes 49. The score of the game is 44.
  Testing objectives: TB, DR7
- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 4. The score at frame #2 is 29. The score at frame #3 becomes 47 (the strike bonus is added). The score at frame #4 becomes 55. The score of the game is 55.
  Testing objectives: TB, DR7
- The game is not finished.
  Testing objectives: DR1

S8: A complete game with regular, non-terminal strike and non-terminal spare frames



- A new game G8 is started.
- A throw knocks down 8 pins (representative case of a first regular throw). The score at frame #1 is 8 and the score of the game is 8.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 0 pins (limit case of a second regular throw). The score at frame #1 is 8 and the score of the game is 8.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 3 pins (representative case of a first regular throw). The score at frame #1 is 8. The score at frame #2 is 11. The score of the game is 11.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 7 pins (representative case of a spare). This is a spare. The score at frame #1 is 8. The score at frame #2 is 18 (the spare bonus cannot be added until the next throw score is known). The score of the game is 18.
  Testing objectives: TB, DR4
- A throw knocks down 10 pins (strike case). This is a strike. The score at frame #1 is 8. The score at frame #2 becomes 28 (the spare bonus is added). The score at frame #3 is 38 (the strike bonus cannot be added until the score of the next throws is known). The score of the game is 38.
  Testing objectives: TB, DR5,DR6, DR7

- A throw knocks down 3 pins (representative case of a first regular throw). The score at frame #1 is 8. he score at frame #2 is 28. The score at frame #3 becomes 41 (the strike bonus is added).The score at frame #4 is 44. The score of the game is 44.
  Testing objectives: TB, DR7

- A throw knocks down 7 pins (representative case of a spare). This is a spare. The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 becomes 48 (the strike bonus is added). The score at frame #4 is 58 (the spare bonus cannot be added until the next throw score is known). The score of the game is 58.
  Testing objectives: TB, DR4, DR7

- A throw knocks down 10 pins (strike case). This is a strike. The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 becomes 68 (the spare bonus is added). The score at frame #5 is 78 (the strike bonus cannot be added until the score of the next throws is known). The score of the game is 78.
  Testing objectives: TB, DR5,DR6

- A throw knocks down 5 pins (representative case of a first regular throw). The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 becomes 83 (the strike bonus is added). The score at frame #6 is 88. The score of the game is 88.
  Testing objectives: TB, DR7

- A throw knocks down 2 pins (representative case of a second regular throw). The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 becomes 85 (the strike bonus is added). The score at frame #6 becomes 92. The score of the game is 92.
  Testing objectives: TB, DR7

- A throw knocks down 7 pins (representative case of a first regular throw). The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 is 85. The score at frame #6 is 92. The score at frame #7 is 99.The score of the game is 99.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 1 pin (representative case of a second regular throw). The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 is 85. The score at frame #6 is 92. The score at frame #7 is 100.The score of the game is 100.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 0 pins (representative case of a first regular throw). The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 is 85. The score at frame #6 is 92. The score at frame #7 is 100. The score at frame #8 is 100. The score of the game is 100.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 10 pins (limit case of a spare). This is a spare.The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 is 85. The score at frame #6 is 92. The score at frame #7 is 100. The score at frame #8 is 110 (the spare bonus cannot be added until the next throw score is known). The score of the game is 110.
  Testing objectives: TB, DR4

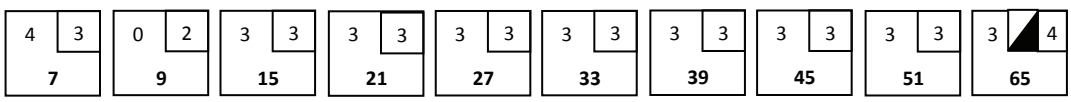
- A throw knocks down 10 pins (strike case). This is a strike. The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 is 85. The score at frame #6 is 92. The score at frame #7 is 100. The score at frame #8 becomes 120 (the spare bonus is added). The score at frame #9 is 130 (the strike bonus cannot be added until the score of the next throws is known). The score of the game is 130.
  Testing objectives: TB, DR6

- A throw knocks down 2 pins (representative case of a first regular throw). The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 is 85. The score at frame #6 is 92. The score at frame #7 is 100. The score at frame #8 is 120. The score at frame #9 becomes 132 (the strike bonus is added). The score at frame #10 is 134.The score of the game is 134.
  Testing objectives: TB, DR7

- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 8. The score at frame #2 is 28. The score at frame #3 is 48. The score at frame #4 is 68. The score at frame #5 is 85. The score at frame #6 is 92. The score at frame #7 is 100. The score at frame #8 is 120. The score at frame #9 becomes 135 (the strike bonus is added). The score at frame #10 becomes 140. The score of the game is 140.
  Testing objectives: TB, DR7

- The game is finished.
  Testing objectives: DR1

S9: A complete game with a spare in the last frame



- A new game G9 is started.

- A throw knocks down 4 pins (representative case of a first regular throw). The score at frame #1 is 4 and the score of the game is 4.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 7 and the score of the game is 7.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 0 pins (limit case of a first regular throw). The score at frame #1 is 7. The score at frame #2 is 7 and the score of the game is 7.
- **Testing objectives:** TB, DR2, DR3

- A throw knocks down 2 pins (representative case of a second regular throw). The score at frame #1 is 7. The score at frame #2 is 9 and the score of the game is 9.
  Testing objectives: TB, DR2, DR3

- 14 regular throws are registered. 3 frames (representative case of a regular throw) are knocked at each throw.
  Testing objectives: TB, DR1, DR2, DR3

- The game is not finished.
  Testing objectives: DR1

- A throw knocks down 3 pins (representative case of a first regular throw). The score at frame #10 is 54 and the score of the game is 54.
  Testing objectives: TB, DR2, DR3

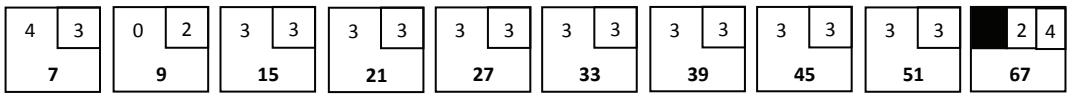
- A throw knocks down 7 pins (representative case of a terminal spare). The score at frame #10 is 61 and the score of the game is 61.
  Testing objectives: TB, DR4, DR8

- The game is not finished.
  Testing objectives: DR1

- A throw knocks down 4 pins (representative case of an additional throw after a terminal spare). The score at frame #10 becomes 65 (the spare bonus is added) and the score of the game is 65.
  Testing objectives: TB, DR5, DR8

- The game is finished.
  Testing objectives: DR1

S10: A complete game with a strike in the last frame



- A new game G10 is started.

- A throw knocks down 4 pins (representative case of a first regular throw). The score at frame #1 is 4 and the score of the game is 4.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 3 pins (representative case of a second regular throw). The score at frame #1 is 7 and the score of the game is 7.
  Testing objectives: TB, DR2, DR3

- A throw knocks down 0 pins (limit case of a first regular throw). The score at frame #1 is 7. The score at frame #2 is 7 and the score of the game is 7.
- **Testing objectives:** TB, DR2, DR3

- A throw knocks down 2 pins (representative case of a second regular throw). The score at frame #1 is 7. The score at frame #2 is 9 and the score of the game is 9.
  Testing objectives: TB, DR2, DR3

- 14 regular throws are registered. 3 frames (representative case of a regular throw) are knocked at each throw.
  Testing objectives: TB, DR1, DR2, DR3

- The game is not finished.
  Testing objectives: DR1

- A throw knocks down 10 pins (representative case of a terminal strike). The score at frame #10 is 61 and the score of the game is 61.
  Testing objectives: TB, DR6

- The game is not finished.
  Testing objectives: DR1

- A throw knocks down 2 pins (representative case of an additional throw after a terminal strike). The score at frame #10 becomes 63 (the strike bonus is added) and the score of the game is 63.
  Testing objectives: TB, DR7, DR8

- The game is not finished.
  Testing objectives: DR1
- A throw knocks down 4 pins (representative case of an additional throw after a terminal strike). The score at frame #10 becomes 67 (the strike bonus is added) and the score of the game is 67.
  Testing objectives: TB, DR7, DR8

- The game is finished.
  Testing objectives: DR1

S11: Throws are not allowed for finished games

This story extends S1 as follows:

- A throw that knocks down 4 pins (representative case of a first regular throw) cannot occur.
  Testing objectives: TB (precondition)

3. TDCM application to the case study

In this section we report the results of the TDCM application in the bowling case study, according to the testing strategy defined in Section 2.

The test processor prototype has been extended in order to automatically collect information about the execution of the test cases and the evolution of the conceptual schema during each TDCM iteration.

We report the following information for each iteration:

- **Input conceptual schema**: It is the initial schema to be evolved in the current iteration. In the first iteration, the input conceptual schema is empty. In the next iterations, the input conceptual schema is the resultant schema of the previous iteration.
- **Iteration objective:** It is the objective to be achieved in the current iteration. According to the testing strategy, an iteration objective corresponds to a (fragment of a) user story that should be feasible in the system under development.
- **Current test case:** It is the iteration objective story written in a formal and executable test case. We use the Conceptual Schema Testing Language (CSTL) [6] to define test cases.
- **Regression test cases:** They are test cases which have been processed in previous iterations (their verdict is *Pass*). We execute them in each iteration to be sure that their verdict remains *Pass* after the changes performed in the current iteration.
- **Test-Driven evolution of the Conceptual Schema Under Development (CSUD):** It is a log of the errors and failures obtained by the test processor prototype during the execution of the tests in the current iteration. In TDCM, failures and errors drive the changes to be done in the schema. These changes are also summarized.
- **Resultant conceptual schema:** It is the input schema with the changes that have been added during the current iteration by applying TDCM. The resultant conceptual schema contains the knowledge to make the processed tests pass.
- **Additional information:** We collect the total time spent to pass the current test case. We also analyze and categorize the errors and failures reported in the iteration and the type of changes applied to fix them. The categorization of errors and the corresponding changes assume that the test cases are syntactically correct and that they correctly formalize the expectations of the user stories.

Iteration 1

Input conceptual schema

The conceptual schema under development (CSUD) is empty .

Iteration objective

S1: A complete game without spares and strike (first fragment).

Current Test case

```
testprogram CompleteGameWithoutSparesAndStrikes{
 test S1{
    ng1 := new NewGame;
    assert occurrence ng1;
    g1 := ng1.createdGame;
    assert true g1.frame->size()=10;

    // The rest of the test case is not considered in this iteration
 }
}
```

Regression test cases

There are no regression test cases.

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- **Current Test Case. Line 4**: *NewGame* is not defined in the CSUD as an entity or a relationship type.

▶ The conceptual modeler defines the event type *NewGame.*

```
event NewGame
operations
effect()
end

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew())
```

- **CSUD**: *Game* is not defined in the CSUD as an entity or a relationship type.

▶ The conceptual modeler defines the entity type *Game*.

- **Current Test Case. Line 5**: The effect of *NewGame* is not defined as a method.

- The conceptual modeler specifies the method of the event type *NewGame.*

```
method NewGame{
g := new Game;
}
```

- **Current Test Case. Line 6**: Undefined property named *createdGame* in expression {NewGame}.createdGame.

- The conceptual modeler adds the property *createdGame*

```
event NewGame
attributes
createdGame:Game
operations
effect()
end
```

- **Current Test Case. Line 5**: Inconsistent state before ng1:NewGame event execution: Instances of NewGame violate the multiplicity createdGame[1].

- The conceptual modeler changes the multiplicity, because the instances of NewGame only have a created game after its occurrence.

```
event NewGame
attributes
createdGame:Game
operations
effect()
```

- **Current Test Case. Line 7**: Undefined property named *frame* in expression {Game}.frame

- The conceptual modeler adds knowledge about the frames of a game.

```
class Frame
end

association game_frame between
  Game[1]
  Frame[10]
End
```

- **Current Test Case. Line 5**: Inconsistent state after ng1:NewGame event execution: Multiplicity constraint violation in association *game_frame*: objects of *Game* are connected to 0 objects of class *Frame* but the multiplicity is specified as 10.

- The conceptual modeler changes the postcondition of the event *NewGame*.

```
context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew() and g.frame->size()=10)
```

- **CSUT**: The postcondition of *NewGame* evaluates to false.

- The conceptual modeler changes the method of *NewGame* to satisfy the postcondition.

```
method NewGame{
g := new Game;
index:=0;
        while index<10 do
```

```
            f:=new Frame(game:=g);
            index:=index+1;
         endwhile
}
```

- **Current test case. Line 7**: Undefined property named *frame* in expression {OclVoid}.frame

▶ The conceptual modeler realizes that the variable *g1* is void. Then, *ng1.createdGame* is empty. The modeler changes the effect postcondition and the method of the event type *NewGame*, in order to know the created game.

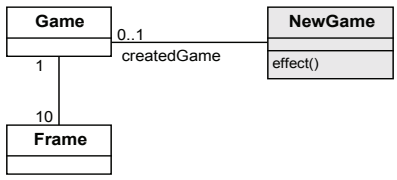
```
context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew() and self.createdGame=g and g.frame->size()=10)

method NewGame{
g := new Game;
index:=0;
                while index<10 do
                  f:=new Frame(game:=g);
                  index:=index+1;
                endwhile

self.createdGame:=g;
}
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema




```
context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew() and self.createdGame=g and g.frame->size()=10)
```

Additional information

TIME TO FINISH THE ITERATION	13 MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD	A derived type involved in a test case does not exist in the CSUD	An event type involved in a test case does not exist in the CSUD
The basic type is relevant and it is added to the CSUD	The derived type is relevant and it is added to the CSUD	The event type is relevant and it is added to the CSUD
3		**1**

Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
1		1		2	

An assertion about the IB state fails or contains an error		**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect	A precondition is added/updated	The expression is corrected	The CSUD is changed
1				

Iteration 2

Input conceptual schema

The resultant conceptual schema of Iteration 1.

Iteration objective

S1: A complete game without spares and strike.

Current Test case

```
testprogram CompleteGameWithoutSparesAndStrikes{
 test S1{
    ng1 := new NewGame;
    assert occurrence ng1;
    g1 := ng1.createdGame;

    assert true g1.frame->size()=10;
    t1 := new NewThrow(knockedPins:=4, game:=g1);
    assert occurrence t1;
    assert equals g1.frame->at(1).score() 4;
    assert equals g1.score() 4;

    t2 := new NewThrow(knockedPins:=3, game:=g1);
    assert occurrence t2;
    assert equals g1.frame->at(1).score() 7;
    assert equals g1.score() 7;

    assert true g1.finished()=false;

    t3 := new NewThrow(knockedPins:=0, game:=g1);
    assert occurrence t3;
    assert equals g1.frame->at(1).score() 7;
    assert equals g1.frame->at(2).score() 7;
    assert equals g1.score() 7;

    t4 := new NewThrow(knockedPins:=2, game:=g1);
    assert occurrence t4;
```

```
    assert equals g1.frame->at(1).score() 7;
    assert equals g1.frame->at(2).score() 9;
    assert equals g1.score() 9;

    index:=0;
    expectedGameScore:=9;
    while index<16 do
      tx:=new NewThrow(knockedPins:=3, game:=g1);
       assert occurrence tx;
      index:=index+1;
      expectedGameScore:=expectedGameScore+3;
       assert equals g1.score() expectedGameScore;
    endwhile

    assert true g1.finished()=true;

  }
}
```

Regression test cases

There are no regression test cases.

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- **Current Test Case. Line 9**: *NewThrow* is not defined in the CSUD as an entity or a relationship type.

▶ The conceptual modeler defines the event type *NewThrow.*

```
event NewThrow
operations
effect()
end

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew() and t.frame.game=game)
```

- **CSUD**: *Throw* is not defined in the CSUD as an entity or a relationship type.

▶ The conceptual modeler defines the entity type *Throw.*

```
class Throw
end
```

- **CSUD**: Undefined property named *frame* in expression {Throw}.frame

▶ The conceptual modeler adds the relationship type between *Frame* and *Throw.*

```
association frame_throw between
  Frame[1]
  Throw[0..2]
end
```

- **CSUD**: Undefined property named *game* in expression {NewThrow}.game

- The conceptual modeler adds the property *game* to the *NewThrow* event.

```
association newThrow_game between
  NewThrow[*]
  Game[1]
end
```

- **Current Test Case. Line 9**: Binary property *knockedPins* does not exist.

- The conceptual modeler adds the property *knockedPins* to the *NewThrow* event.

```
event NewThrow
attributes
 knockedPins:Integer
operations
effect()
end
```

- **Current Test Case. Line 10**: The effect of *NewThrow* is not defined as a method. Specify/review first its postconditions.

- The conceptual modeler adds the method of *NewThrow.*

```
method NewThrow{
 t:=new Throw;
 t.frame:=self.game.frame->select(f|f.throw->size()<2)->first();
}
```

- **Method**: Undefined operation Set(Frame)->at(Integer).

- The conceptual modeler realizes that the frames of a game need to be ordered.

```
association game_frame between
  Game[1]
  Frame[10] ordered
End
```

- **Current Test Case. Line 7**: Undefined property named *score* in expression {Frame}.score

- The conceptual modeler adds knowledge about the score of a game.

```
class Frame
attributes
/score:Integer=self.throw.knockedPins->sum()
End
```

- **CSUD**: Undefined property *knockedPins* for the entity type *Throw*.

- The conceptual modeler adds the property *Throw::KnockedPins.*

```
class Throw
attributes
knockedPins:Integer
end
```

- **Current Test case. Line 10**: Inconsistent state after t1:NewThrow event execution. Instances of Throw violate the multiplicity knockedPins[1].

- The conceptual modeler realizes that the knocked pins are not assigned to the entity type *Throw*. The postcondition and the method of the event type *NewThrow* are extended.

```
context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew() and t.frame.game=game and t.knockedPins=self.knockedPins)


method NewThrow{
 t:=new Throw(knockedPins:=self.knockedPins);
 t.frame:=self.game.frame->select(f|f.throw->size()<2)->first();
}
```

- **Current Test case. Line 12**: Undefined property named *score* in expression {Game}.score

▶ The conceptual modeler realizes that the knowledge about the score of a game does not exist in the CSUD.

```
class Game
operations
score():Integer=self.frame.score()->sum()
end
```

- **Current Test case. Line 19**: Undefined property named *finished* in expression {Game}.finished

▶ The conceptual modeler adds the derived attribute *Game::finished*

```
class Game
operations
score():Integer=self.frame.score()->sum()
finished():Integer=self.frame.throw->size()=20
end
```

- **Test case. Line 24**: The result of *g1.frame->at(2).score()* is 0 but it is expected to be 7.

▶ The conceptual modeler realizes that the user story assumes that the score in a frame is accumulative (it is the number of knocked pins in the current frame, adding the score at the previous frame).

```
class Frame
operations
score():Integer= if self.previous().isUndefined()
                 then self.throw.knockedPins->sum()
                 else self.throw.knockedPins->sum()+self.previous().score()
                 endif
end
```

- **CSUD**: Undefined property named *previous* in expression {Frame}.previous

▶ The conceptual modeler needs to specify the knowledge about the previous frame of a frame.

```
class Frame
operations
…
previous():Frame=self.game.frame->any(f|f.frameNumber=self.frameNumber-1)
end
```

- **CSUD**: Undefined property named *frameNumber* in expression {Frame}.frameNumber

▶ The conceptual modeler needs to specify the number of each frame.

```
class Frame
attributes
frameNumber:Integer
```

```
…
end
```

- **Current test case. Line 5.**: Inconsistent state after ng1:NewGame event execution: Instances of frame violate the multiplicity *Frame::frameNumber*.

▶ The conceptual modeler realizes that the effect of the event type *NewGame* needs to include the number of the created frames.

```
context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10})

method NewGame{
g := new Game;
index:=0;
  while index<10 do
   f:=new Frame(game:=g);
   index:=index+1;
  endwhile
index:=0;
   while index<10 do
     f:=g.frame->at(index+1);
     f.frameNumber:=index+1;
     index:=index+1;
   endwhile

self.createdGame:=g;
}
```

- **Current test case. Line 12**: The result of *g1.score()* is 40 but it is expected to be 4.

▶ The conceptual modeler realizes that the score of the game is not the sum of the score of all the frames, but the score of the active frame.

```
class Game
operations
score():Integer=if self.activeFrame->isEmpty()
                then 0
                else self.activeFrame.score()
                endif
…
end
```

- **CSUD**: Undefined property *activeFrame* in expression Game.activeFrame

▶ The conceptual modeler changes the schema to include the active frame of a game.

```
association game_activeFrame between
  Game[1] role gameOfActiveFrame
  Frame[1] role activeFrame
End
```

- **Current test case. Line 5**: Inconsistent state after ng1:NewGame event execution: Multiplicity constraint violation in association *game_activeFrame.* Object of class *Game* is connected to 0 objects of class *Frame,* but the multiplicity is specified as 1.

▶ The conceptual modeler identifies that a new throw determines the active frame of a game. Then, the modeler changes the postcondition of the event type *NewThrow* in order to include this knowledge.

```
context NewThrow::effect()
post:
```

```
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.frame.game=game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game.frame@pre->select(f|f.throw->size()<2)->first()
             and t.game.activeFrame=t.frame)
```

- **CSUD**: Undefined property named *game* in expression {Throw}.game

▶ The conceptual modeler changes the schema in order to include the game of a throw.

```
association throw_game between
  Throw[0..20]
  Game[1]
End
```

- **Current test case. Line 5**: Inconsistent state after ng1:NewGame event execution: Multiplicity constraint violation in association *game_activeFrame.* Object of class *Game* is connected to 0 objects of class *Frame,* but the multiplicity is specified as 1.

▶ The conceptual modeler realizes that although a new throw determines the active frame, the event type *NewGame* event needs to initialize the active frame when a new game is created.

```
context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
             and g.activeFrame=g.frame->sortedBy(frameNumber)->first())


method NewGame{
…
g.activeFrame:=g.frame->first();
}
```

- **Current test case. Line 5**: Inconsistent state after ng1:NewGame event execution: Multiplicity constraint violation in association *game_activeFrame.* Object of class *Game* is connected to 0 objects of class *Frame,* but the multiplicity is specified as 1.

▶ At this point, the conceptual modeler realizes that the multiplicity of the association game_activeFrame is incorrect.

```
association game_activeFrame between
  Game[0..1] role gameOfActiveFrame
  Frame[1] role activeFrame
End
```

- **CSUD**: The postcondition of the effect of the event *NewThrow* evaluates to false. The expression *t.game* evaluates to *Undefined*.

▶ The conceptual modeler realizes that the event type *NewThrow* does not define the game of a throw.

```
method NewThrow{
 t:=new Throw(knockedPins:=self.knockedPins);
 t.game:=self.game;
 t.frame:=self.game.frame->sortedBy(frameNumber)->select(f|f.throw->size()<2)->first();
 t.frame.game.activeFrame:=t.frame;
}
```

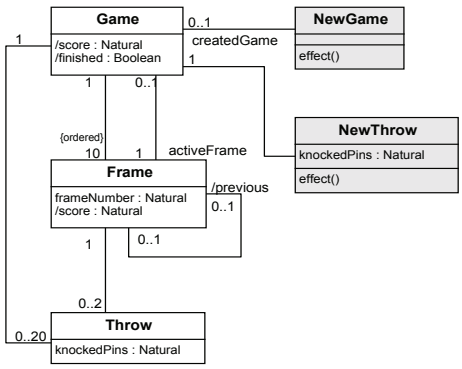
- **CSUD**: The postcondition of the effect of the event *NewThrow* evaluates to false. The expression *t.frame=t.game.frame->select(f | f.throw ->size()<2)->first()* evaluates to false.

▶ The conceptual modeler corrects the postcondition.

```
context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game.frame->select(f|f.throw@pre->size()<2)->first()
             and t.game.activeFrame=t.frame)
```

- The verdict of the current test set is *Pass*.

The resultant conceptual schema is:




```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
               then 0
               else self.activeFrame.score
               endif

context Game::finished:Boolean
  derive: self.frame.throw->size()=20

context Frame::score:Integer
  derive: if self.previous.isUndefined()
               then self.throw.knockedPins->sum()
               else self.throw.knockedPins->sum()+self.previous.score
               endif

context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
             and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game.frame->select(f|f.throw@pre->size()<2)->first()
             and t.game.activeFrame=t.frame)
```

- The conceptual modeler observes that there is a cycle. The conceptual modeler refactors the schema because the game of a throw may be derived from its frame:

```
class Throw
attributes
knockedPins:Integer
operations
game():Game=self.frame.game
end
```

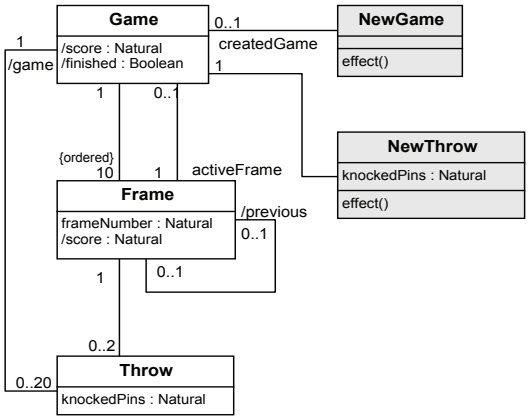
- **CSUD**: Binary property game is derived. No links can be explicitly created.

- The conceptual modeler deletes the expression t.game:=self.game in the method.

```
method NewThrow{
t:=new Throw(knockedPins:=self.knockedPins);
~~t.game:=self.game;~~
t.frame:=self.game.frame->sortedBy(frameNumber)->select(f|f.throw->size()<2)->first();
t.frame.game.activeFrame:=t.frame;
}
```

- The verdict of the current test set remains *Pass*.

Resultant conceptual schema




```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
                then 0
                else self.activeFrame.score
                endif

context Game::finished:Boolean
  derive: self.frame.throw->size()=20

context Frame::score:Integer
  derive: if self.previous.isUndefined()
                then self.throw.knockedPins->sum()
                else self.throw.knockedPins->sum()+self.previous.score
                endif

context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)

context Throw::game:Game
  derive: self.frame.game
```

```
context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
             and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
              and t.game=self.game
              and t.knockedPins=self.knockedPins
              and t.frame=t.game.frame->select(f|f.throw@pre->size()<2)->first()
              and t.game.activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	38 MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD		**A derived type involved in a test case does not exist in the CSUD**		**An event type involved in a test case does not exist in the CSUD**	
The basic type is relevant and it is added to the CSUD		The derived type is relevant and it is added to the CSUD		The event type is relevant and it is added to the CSUD	
9		4		1	
Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
		4	1		1
An assertion about the IB state fails or contains an error			**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect		A precondition is added/updated	The expression is corrected	The CSUD is changed
	2				2

Iteration 3

Input conceptual schema

The resultant conceptual schema of Iteration 2.

Iteration objective

S2: An incomplete game with a spare in the first frame

Current Test case

```
testprogram IncompleteGameWithASpareInTheFirstFrame{

 test S2{
    ng2 := new NewGame;
```

```
    assert occurrence ng2;
    g2 := ng2.createdGame;

    t1 := new NewThrow(knockedPins:=4, game:=g2);
    assert occurrence t1;
    assert equals g2.frame->at(1).score() 4;
    assert equals g2.score() 4;

    t2 := new NewThrow(knockedPins:=6, game:=g2);
    assert occurrence t2;
    assert true g2.frame->at(1).isSpare();
    assert equals g2.frame->at(1).score() 10;
    assert equals g2.score() 10;

    t3 := new NewThrow(knockedPins:=7, game:=g2);
    assert occurrence t3;
    assert equals g2.frame->at(1).score() 17;
    assert equals g2.frame->at(2).score() 24;
    assert equals g2.score() 24;

    assert true g2.finished()=false;
}

}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- ● **Current Test Case. Line 15**: Undefined property named *isSpare* in expression Frame.isSpare
- ▶ The conceptual modeler adds the derived attribute *Frame::isSpare*

```
class Frame
attributes
frameNumber:Integer
operations
isSpare():Boolean=self.throw->size()>1
                  and self.throw->at(1).knockedPins  + self.throw->at(2).knockedPins=10
…
end
```

- ● **CSUD**: Undefined operation Set(Throw)->at(Integer) in expression *self.throw->at(1).knockedPins*
- ▶ The conceptual modeler realizes that the throws of a frame need to be ordered.

```
association frame_throw between
  Frame[1]
  Throw[0..2] ordered
end
```

- ● **Current Test Case. Line 21**: The result is 10 but it is expected to be 17 (assertion: *assert equals g2.frame->at(1).score() 17*).

- The conceptual modeler modifies the derived attribute *score* in order to deal with spare frames (12 executions have been done in order to reach this derivation rule without syntax errors and without altering the verdict of the previous test program).

```
attributes
frameNumber:Integer
operations
…
score():Integer=
                let knockedPins:Integer=
                        self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->at(1).isDefined() then
                        if self.isSpare() then self.next().throw->at(1).knockedPins else 0
endif
                       else 0
                       endif
                in
                    knockedPins+spareBonus+scoreAtPreviousFrame
…
end
```

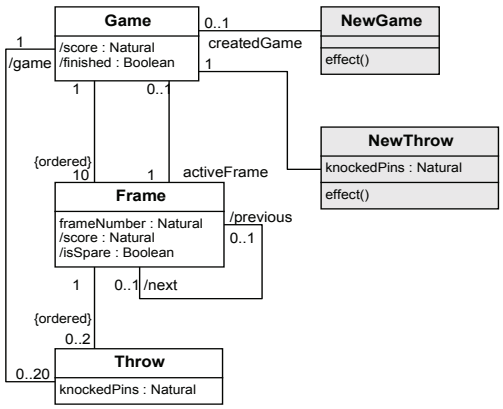
- **CSUD**: Undefined property named *next* in expression {Frame}.next

- The conceptual modeler adds the derivation rule to the association end *Frame::next*

```
class Frame
…
operations
…
previous():Frame=self.game.frame->any(f|f.frameNumber=self.frameNumber-1)
next():Frame=self.game.frame->any(f|f.frameNumber=self.frameNumber+1)
end
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema



```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
               then 0
               else self.activeFrame.score
               endif

context Game::finished:Boolean
  derive: self.frame.throw->size()=20

context Frame::isSpare:Boolean
  derive: self.throw->size()>1
          and self.throw->at(1).knockedPins  + self.throw->at(2).knockedPins=10

context Frame::score:Integer
  derive: let knockedPins:Integer=
                       self.throw.knockedPins->sum()
               in
               let scoreAtPreviousFrame=
                      if self.previous().isUndefined() then 0
                      else
                       self.previous().score()
                      endif
               in
               let spareBonus=
                      if self.next().throw->at(1).isDefined() then
                          if self.isSpare() then
                             self.next().throw->at(1).knockedPins
                          else 0
                          endif
                      else 0
                      endif
               in
                   knockedPins+spareBonus+scoreAtPreviousFrame


context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)


context Frame::next:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber+1)


context Throw::game:Game
  derive: self.frame.game


context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
             and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
              and t.game=self.game
              and t.knockedPins=self.knockedPins
              and t.frame=t.game.frame->select(f|f.throw@pre->size()<2)->first()
              and t.game.activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	23 MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD		**A derived type involved in a test case does not exist in the CSUD**		**An event type involved in a test case does not exist in the CSUD**	
The basic type is relevant and it is added to the CSUD		The derived type is relevant and it is added to the CSUD		The event type is relevant and it is added to the CSUD	
		2			
Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
		4	**1**		**1**
An assertion about the IB state fails or contains an error			**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect		A precondition is added/updated	The expression is corrected	The CSUD is changed
	1				**1**

Iteration 4

Input conceptual schema

The resultant conceptual schema of Iteration 3.

Iteration objective

S3: An incomplete game with regular and non-terminal spare frames

Current Test case

```
testprogram IncompleteGameWithRegularAndNonTerminalSpareFrames{

 test S3{
   ng3 := new NewGame;
    assert occurrence ng3;
    g3 := ng3.createdGame;

    t1 := new NewThrow(knockedPins:=0, game:=g3);
    assert occurrence t1;
    assert equals g3.frame->at(1).score() 0;
    assert equals g3.score() 0;

    t2 := new NewThrow(knockedPins:=0, game:=g3);
    assert occurrence t2;
    assert equals g3.frame->at(1).score() 0;
    assert equals g3.score() 0;

    t3 := new NewThrow(knockedPins:=0, game:=g3);
    assert occurrence t3;
    assert equals g3.frame->at(1).score() 0;
```

```
assert equals g3.frame->at(2).score() 0;
assert equals g3.score() 0;

t4 := new NewThrow(knockedPins:=3, game:=g3);
assert occurrence t4;
assert equals g3.frame->at(1).score() 0;
assert equals g3.frame->at(2).score() 3;
assert equals g3.score() 3;

t5 := new NewThrow(knockedPins:=4, game:=g3);
assert occurrence t5;
assert equals g3.frame->at(1).score() 0;
assert equals g3.frame->at(2).score() 3;
assert equals g3.frame->at(3).score() 7;
assert equals g3.score() 7;

t6 := new NewThrow(knockedPins:=5, game:=g3);
assert occurrence t6;
assert equals g3.frame->at(1).score() 0;
assert equals g3.frame->at(2).score() 3;
assert equals g3.frame->at(3).score() 12;
assert equals g3.score() 12;

t7 := new NewThrow(knockedPins:=0, game:=g3);
assert occurrence t7;
assert equals g3.frame->at(1).score() 0;
assert equals g3.frame->at(2).score() 3;
assert equals g3.frame->at(3).score() 12;
assert equals g3.frame->at(4).score() 12;
assert equals g3.score() 12;

t8 := new NewThrow(knockedPins:=10, game:=g3);
assert occurrence t8;
assert equals g3.frame->at(1).score() 0;
assert equals g3.frame->at(2).score() 3;
assert equals g3.frame->at(3).score() 12;
assert equals g3.frame->at(4).score() 22;
assert true g3.frame->at(4).isSpare();
assert equals g3.score() 22;

t9 := new NewThrow(knockedPins:=6, game:=g3);
assert occurrence t9;
assert equals g3.frame->at(1).score() 0;
assert equals g3.frame->at(2).score() 3;
assert equals g3.frame->at(3).score() 12;
assert equals g3.frame->at(4).score() 28;
assert equals g3.frame->at(5).score() 34;
assert equals g3.score() 34;

t10 := new NewThrow(knockedPins:=3, game:=g3);
assert occurrence t10;
assert equals g3.frame->at(1).score() 0;
assert equals g3.frame->at(2).score() 3;
assert equals g3.frame->at(3).score() 12;
assert equals g3.frame->at(4).score() 28;
assert equals g3.frame->at(5).score() 37;
assert equals g3.score() 37;

t11 := new NewThrow(knockedPins:=3, game:=g3);
```

```
    assert occurrence t11;
    assert equals g3.frame->at(1).score() 0;
    assert equals g3.frame->at(2).score() 3;
    assert equals g3.frame->at(3).score() 12;
    assert equals g3.frame->at(4).score() 28;
    assert equals g3.frame->at(5).score() 37;
    assert equals g3.frame->at(6).score() 40;
    assert equals g3.score() 40;

    t12 := new NewThrow(knockedPins:=7, game:=g3);
    assert occurrence t12;
    assert equals g3.frame->at(1).score() 0;
    assert equals g3.frame->at(2).score() 3;
    assert equals g3.frame->at(3).score() 12;
    assert equals g3.frame->at(4).score() 28;
    assert equals g3.frame->at(5).score() 37;
    assert true g3.frame->at(6).isSpare();
    assert equals g3.frame->at(6).score() 47;
    assert equals g3.score() 47;

    t13 := new NewThrow(knockedPins:=0, game:=g3);
    assert occurrence t13;
    assert equals g3.frame->at(1).score() 0;
    assert equals g3.frame->at(2).score() 3;
    assert equals g3.frame->at(3).score() 12;
    assert equals g3.frame->at(4).score() 28;
    assert equals g3.frame->at(5).score() 37;
    assert equals g3.frame->at(6).score() 47;
    assert equals g3.frame->at(7).score() 47;
    assert equals g3.score() 47;

    assert true g3.finished()=false;
}
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- The verdict of the current test set is *Pass*.

Resultant conceptual schema

No changes have been done in the CSUD in this iteration.

Additional information

TIME TO FINISH THE ITERATION	35 SEG

Iteration 5

Input conceptual schema

The resultant conceptual schema of Iteration 4.

Iteration objective

S4: An incomplete game with two consecutive and non-terminal spares

Current Test case

```
testprogram IncompleteGameWithTwoConsecutiveAndNonTerminalSpares{

 test S4{
   ng4 := new NewGame;
   assert occurrence ng4;
   g4 := ng4.createdGame;

   t1 := new NewThrow(knockedPins:=1, game:=g4);
   assert occurrence t1;
   assert equals g4.frame->at(1).score() 1;
   assert equals g4.score() 1;

   t2 := new NewThrow(knockedPins:=2, game:=g4);
   assert occurrence t2;
   assert equals g4.frame->at(1).score() 3;
   assert equals g4.score() 3;

   t3 := new NewThrow(knockedPins:=2, game:=g4);
   assert occurrence t3;
   assert equals g4.frame->at(1).score() 3;
   assert equals g4.frame->at(2).score() 5;
   assert equals g4.score() 5;

   t4 := new NewThrow(knockedPins:=8, game:=g4);
   assert occurrence t4;
   assert equals g4.frame->at(1).score() 3;
   assert true g4.frame->at(2).isSpare();
   assert equals g4.frame->at(2).score() 13;
   assert equals g4.score() 13;

   t5 := new NewThrow(knockedPins:=4, game:=g4);
   assert occurrence t5;
   assert equals g4.frame->at(1).score() 3;
   assert equals g4.frame->at(2).score() 17;
   assert equals g4.frame->at(3).score() 21;
   assert equals g4.score() 21;
```

```
  t6 := new NewThrow(knockedPins:=6, game:=g4);
  assert occurrence t6;
  assert equals g4.frame->at(1).score() 3;
  assert equals g4.frame->at(2).score() 17;
  assert equals g4.frame->at(3).score() 27;
  assert true g4.frame->at(3).isSpare();
  assert equals g4.score() 27;

  t7 := new NewThrow(knockedPins:=5, game:=g4);
  assert occurrence t7;
  assert equals g4.frame->at(1).score() 3;
  assert equals g4.frame->at(2).score() 17;
  assert equals g4.frame->at(3).score() 32;
  assert equals g4.frame->at(4).score() 37;
  assert equals g4.score() 37;

  assert true g4.finished()=false;
 }
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)
- IncompleteGameWithRegularAndNonTerminalSpareFrames (S3)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- **Current Test Case. Line 41**: The result is 19 but is expected to be 17 in: a*ssert equals g4.frame->at(2).score() 17;*

▶ The conceptual modeler realizes that the throws of a frame are not ordered as expected. Then, the modeler specifies uses the explicitly specified order when a throw is created.

```
class Frame
attributes
frameNumber:Integer
operations
…
score():Integer=
                let knockedPins:Integer=
                       self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
```

```
                    if self.isSpare() then
                      self.next().throw->sortedBy(order)->at(1).knockedPins
                    else 0
                    endif
                  else 0
                  endif
in
    knockedPins+spareBonus+scoreAtPreviousFrame
```

```
context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->select(f|f.throw@pre->size()<2)->first()
             and t.order=t.frame.throw->size()
             and t.game().activeFrame=t.frame)
```

- **CSUT**: Undefined property named *order* in expression {Throw}.order

▶ The order of a throw is specified.

```
class Throw
attributes
  knockedPins:Integer
  order:Integer
…
end
```

- **CompleteGameWithoutSparesAndStrikes. Line 10.**: Inconsistent state after t1:NewThrow event execution: Instances of Throw violate the multiplicity Throw::order[1]

▶ The order of a throw within a frame is not specified when the order is created.

```
context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->select(f|f.throw@pre->size()<2)->first()
             and t.order=t.frame.throw->size()
             and t.game().activeFrame=t.frame)
```

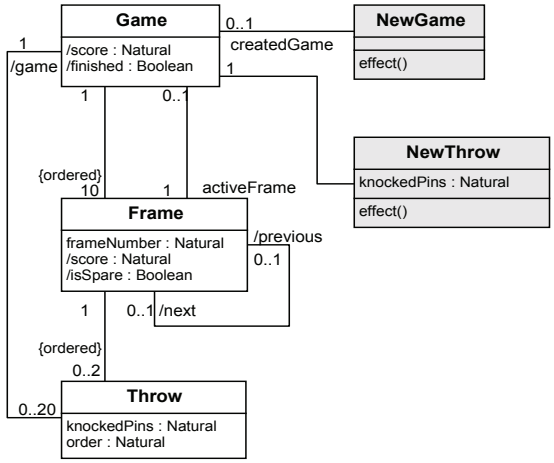
- **CSUT**: The postcondition of NewThrow is not satisfied.

▶ The method of the effect of NewThrow needs to be changed according to the postcondition.

```
method NewThrow{
 t:=new Throw(knockedPins:=self.knockedPins);
 t.frame:=self.game.frame->sortedBy(frameNumber)->select(f|f.throw->size()<2)->first();
 t.order:=t.frame.throw->size();
 t.frame.game.activeFrame:=t.frame;
}
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema




```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
               then 0
               else self.activeFrame.score
               endif

context Game::finished:Boolean
  derive: self.frame.throw->size()=20

context Frame::isSpare:Boolean
  derive: self.throw->size()>1
          and self.throw->at(1).knockedPins  + self.throw->at(2).knockedPins=10

context Frame::score:Integer
  derive: let knockedPins:Integer=
                        self.throw.knockedPins->sum()
               in
               let scoreAtPreviousFrame=
                      if self.previous().isUndefined() then 0
                      else
                       self.previous().score()
                      endif
               in
               let spareBonus=
                      if self.next().throw->sortedBy(order)->at(1).isDefined() then
                         if self.isSpare() then
                             self.next().throw->sortedBy(order)->at(1).knockedPins
                         else 0
                         endif
                      else 0
                      endif
               in
                   knockedPins+spareBonus+scoreAtPreviousFrame

context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)


context Frame::next:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber+1)


context Throw::game:Game
  derive: self.frame.game

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
            and self.createdGame=g
            and g.frame->size()=10
            and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
            and g.activeFrame=g.frame->sortedBy(frameNumber)->first())
```

```
context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->select(f|f.throw@pre->size()<2)->first()
             and t.order=t.frame.throw->size()
              and t.game().activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	16 MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD		**A derived type involved in a test case does not exist in the CSUD**		**An event type involved in a test case does not exist in the CSUD**	
The basic type is relevant and it is added to the CSUD		The derived type is relevant and it is added to the CSUD		The event type is relevant and it is added to the CSUD	
1					
Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
		1		1	

An assertion about the IB state fails or contains an error		**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect	A precondition is added/updated	The expression is corrected	The CSUD is changed
	1			

Iteration 6

Input conceptual schema

The resultant conceptual schema of Iteration 5.

Iteration objective

S5: An incomplete game with a strike in the first frame

Current Test case

```
testprogram IncompleteGameWithStrikeInFirstFrame{

 test S5{
   ng5 := new NewGame;
    assert occurrence ng5;
    g5 := ng5.createdGame;
```

```
  t1 := new NewThrow(knockedPins:=10, game:=g5);
  assert occurrence t1;
  assert true g5.frame->at(1).isStrike();
  assert false g5.frame->at(1).isSpare();
  assert equals g5.frame->at(1).score() 10;
  assert equals g5.score() 10;

  t2 := new NewThrow(knockedPins:=6, game:=g5);
  assert occurrence t2;
  assert equals g5.frame->at(1).score() 16;
  assert equals g5.frame->at(2).score() 22;
  assert equals g5.score() 22;

  t3 := new NewThrow(knockedPins:=2, game:=g5);
  assert occurrence t3;
  assert equals g5.frame->at(1).score() 18;
  assert equals g5.frame->at(2).score() 26;
  assert equals g5.score() 26;

  assert true g5.finished()=false;

 }
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)
- IncompleteGameWithRegularAndNonTerminalSpareFrames (S3)
- IncompleteGameWithTwoConsecutiveAndNonTerminalSpares (S4)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- **Current Test Case. Line 10**: Undefined property *isStrike* in expression Frame.isStrike

▶ The conceptual modeler defines the attribute *Frame::isStrike*.

```
class Frame
attributes
 frameNumber:Integer
operations
  isSpare():Boolean=self.throw->size()>1
                    and self.throw->at(1).knockedPins  + self.throw->at(2).knockedPins=10
  isStrike():Boolean=self.throw->size()=1
                     and self.throw->sortedBy(order)->at(1).knockedPins=10
…
end
```

- **Current Test Case. Line 18**: The result is 16 but it is expected to be 22 in assertion *assert equals g5.frame->at(2).score() 22;*

▶ The conceptual modeler modifies the derived attribute *score* in order to deal with strike frames

```
class Frame
…
score():Integer=
                let knockedPins:Integer=
                        self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                        if self.isSpare() then
                           self.next().throw->sortedBy(order)->at(1).knockedPins
                        else 0
                        endif
                       else 0
                       endif
                in
                let strikeBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                          if self.isStrike() then
                             self.next().throw->sortedBy(order)->at(1).knockedPins
                          else 0
                          endif
                       else 0
                       endif
                    +
                      if self.next().throw->sortedBy(order)->at(2).isDefined() then
                          if self.isStrike() then
                             self.next().throw->sortedBy(order)->at(2).knockedPins
                          else 0
                          endif
                       else 0
                       endif
                in
                    knockedPins+spareBonus+strikeBonus+scoreAtPreviousFrame
```

- **Current Test Case. Line 18**: The result is 16 but it is expected to be 22 in assertion *assert equals g5.frame->at(2).score() 22;*

▶ The conceptual modeler realizes that a throw is not correctly assigned to a frame because a strike frame has only one throw.

```
context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->sortedBy(frameNumber)
                       ->reject(f|f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10)
                       ->first()
             and t.order=t.frame.throw->size()
             and t.game().activeFrame=t.frame)
```

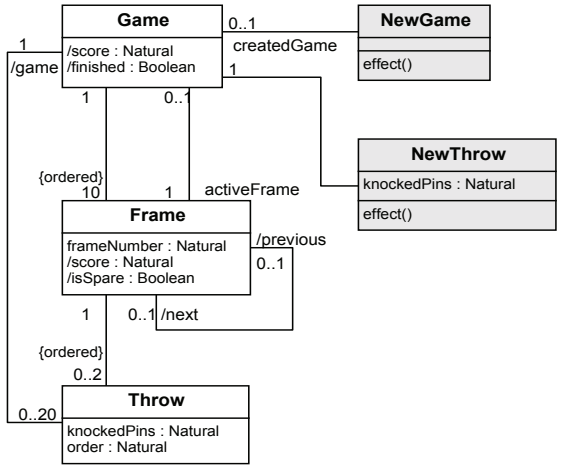
- **CSUT**: The postcondition of *NewThrow* is not satisfied.

▶ The conceptual modeler changes the method of *NewThrow*.

```
method NewThrow{
t:=new Throw(knockedPins:=self.knockedPins);
firstIncompleteFrame:=self.game.frame->sortedBy(frameNumber)
                     ->reject(f|f.finished())->first();
t.frame:=firstIncompleteFrame;
t.order:=t.frame.throw->size();
t.frame.game.activeFrame:=t.frame;
}
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema




```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
               then 0
               else self.activeFrame.score
               endif

context Game::finished:Boolean
  derive: self.frame.throw->size()=20

context Frame::isSpare:Boolean
  derive: self.throw->size()>1
          and self.throw->at(1).knockedPins  + self.throw->at(2).knockedPins=10

context Frame::isStrike:Boolean
  derive: self.throw->size()=1
                    and self.throw->sortedBy(order)->at(1).knockedPins=10

context Frame::score:Integer
  derive:       let knockedPins:Integer=
                        self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                        if self.isSpare() then
                           self.next().throw->sortedBy(order)->at(1).knockedPins
                        else 0
                        endif
                       else 0
                       endif
                in
                let strikeBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                          if self.isStrike() then
                             self.next().throw->sortedBy(order)->at(1).knockedPins
                          else 0
                          endif
                       else 0
                       endif
                    +
                      if self.next().throw->sortedBy(order)->at(2).isDefined() then
                          if self.isStrike() then
                             self.next().throw->sortedBy(order)->at(2).knockedPins
                          else 0
                          endif
                       else 0
                       endif
                in
```

```
                    knockedPins+spareBonus+strikeBonus+scoreAtPreviousFrame

context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)

context Frame::next:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber+1)


context Throw::game:Game
  derive: self.frame.game

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
             and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
              and t.game()=self.game
              and t.knockedPins=self.knockedPins
              and t.frame=t.game().frame->sortedBy(frameNumber)
                        ->reject(f|f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10)
                        ->first()
              and t.order=t.frame.throw->size()
              and t.game().activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	26 MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD		**A derived type involved in a test case does not exist in the CSUD**		**An event type involved in a test case does not exist in the CSUD**	
The basic type is relevant and it is added to the CSUD		The derived type is relevant and it is added to the CSUD		The event type is relevant and it is added to the CSUD	
1					
Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
				1	
An assertion about the IB state fails or contains an error			**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect		A precondition is added/updated	The expression is corrected	The CSUD is changed
1	1				

Iteration 7

Input conceptual schema

The resultant conceptual schema of Iteration 6.

Iteration objective

S6: An incomplete game with regular and non-terminal strike frames

Current Test case

```
testprogram IncompleteGameWithRegularAndNonTerminalStrikeFrames{

 test S6{
   ng6 := new NewGame;
   assert occurrence ng6;
   g6 := ng6.createdGame;

   t1 := new NewThrow(knockedPins:=6, game:=g6);
   assert occurrence t1;
   assert equals g6.frame->at(1).score() 6;
   assert equals g6.score() 6;

   t2 := new NewThrow(knockedPins:=0, game:=g6);
   assert occurrence t2;
   assert equals g6.frame->at(1).score() 6;
   assert equals g6.score() 6;

   t3 := new NewThrow(knockedPins:=2, game:=g6);
   assert occurrence t3;
   assert equals g6.frame->at(1).score() 6;
   assert equals g6.frame->at(2).score() 8;
   assert equals g6.score() 8;
   t4 := new NewThrow(knockedPins:=3, game:=g6);
   assert occurrence t4;
   assert equals g6.frame->at(1).score() 6;
   assert equals g6.frame->at(2).score() 11;
   assert equals g6.score() 11;

   t5 := new NewThrow(knockedPins:=4, game:=g6);
   assert occurrence t5;
   assert equals g6.frame->at(1).score() 6;
   assert equals g6.frame->at(2).score() 11;
   assert equals g6.frame->at(3).score() 15;
   assert equals g6.score() 15;

   t6 := new NewThrow(knockedPins:=5, game:=g6);
   assert occurrence t6;
   assert equals g6.frame->at(1).score() 6;
   assert equals g6.frame->at(2).score() 11;
   assert equals g6.frame->at(3).score() 20;
   assert equals g6.score() 20;
```

t7 := new NewThrow(knockedPins:=10, game:=g6);
assert occurrence t7;
assert true g6.frame->at(4).isStrike();
assert equals g6.frame->at(1).score() 6;
assert equals g6.frame->at(2).score() 11;
assert equals g6.frame->at(3).score() 20;
assert equals g6.frame->at(4).score() 30;
assert equals g6.score() 30;

t8 := new NewThrow(knockedPins:=6, game:=g6);
assert occurrence t8;
assert equals g6.frame->at(1).score() 6;
assert equals g6.frame->at(2).score() 11;
assert equals g6.frame->at(3).score() 20;
assert equals g6.frame->at(4).score() 36;
assert equals g6.frame->at(5).score() 42;
assert equals g6.score() 42;

t9 := new NewThrow(knockedPins:=3, game:=g6);
assert occurrence t9;
assert equals g6.frame->at(1).score() 6;
assert equals g6.frame->at(2).score() 11;
assert equals g6.frame->at(3).score() 20;
assert equals g6.frame->at(4).score() 39;
assert equals g6.frame->at(5).score() 48;
assert equals g6.score() 48;

t10 := new NewThrow(knockedPins:=10, game:=g6);
assert occurrence t10;
assert true g6.frame->at(6).isStrike();
assert equals g6.frame->at(1).score() 6;
assert equals g6.frame->at(2).score() 11;
assert equals g6.frame->at(3).score() 20;
assert equals g6.frame->at(4).score() 39;
assert equals g6.frame->at(5).score() 48;
assert equals g6.frame->at(6).score() 58;
assert equals g6.score() 58;

t11 := new NewThrow(knockedPins:=2, game:=g6);
assert occurrence t11;
assert equals g6.frame->at(1).score() 6;
assert equals g6.frame->at(2).score() 11;
assert equals g6.frame->at(3).score() 20;
assert equals g6.frame->at(4).score() 39;
assert equals g6.frame->at(5).score() 48;
assert equals g6.frame->at(6).score() 60;
assert equals g6.frame->at(7).score() 62;
assert equals g6.score() 62;

t12 := new NewThrow(knockedPins:=2, game:=g6);
assert occurrence t12;
assert equals g6.frame->at(1).score() 6;
assert equals g6.frame->at(2).score() 11;
assert equals g6.frame->at(3).score() 20;
assert equals g6.frame->at(4).score() 39;
assert equals g6.frame->at(5).score() 48;
assert equals g6.frame->at(6).score() 62;
assert equals g6.frame->at(7).score() 66;

```
    assert equals g6.score() 66;

    assert true g6.finished()=false;
}
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)
- IncompleteGameWithRegularAndNonTerminalSpareFrames (S3)
- IncompleteGameWithTwoConsecutiveAndNonTerminalSpares (S4)
- IncompleteGameWithStrikeInFirstFrame (S5)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- The verdict of the current test set is *Pass*.

Resultant conceptual schema

No changes have been done in the CSUD in this iteration.

Additional information

TIME TO FINISH THE ITERATION	28 SEG

Iteration 8

Input conceptual schema

The resultant conceptual schema of Iteration 7.

Iteration objective

S7: An incomplete game with two consecutive and non-terminal strikes

Current Test case

```
testprogram IncompleteGameWithTwoConsecutiveAndNonTerminalStrikes{

 test S7{
   ng7 := new NewGame;
   assert occurrence ng7;
   g7 := ng7.createdGame;

   t1 := new NewThrow(knockedPins:=4, game:=g7);
   assert occurrence t1;
   assert equals g7.frame->at(1).score() 4;
   assert equals g7.score() 4;

   t2 := new NewThrow(knockedPins:=0, game:=g7);
   assert occurrence t2;
   assert equals g7.frame->at(1).score() 4;
   assert equals g7.score() 4;

   t3 := new NewThrow(knockedPins:=10, game:=g7);
   assert occurrence t3;
   assert equals g7.frame->at(1).score() 4;
   assert true g7.frame->at(2).isStrike();
   assert equals g7.frame->at(2).score() 14;
   assert equals g7.score() 14;

   t4 := new NewThrow(knockedPins:=10, game:=g7);
   assert occurrence t4;
   assert equals g7.frame->at(1).score() 4;
   assert true g7.frame->at(2).isStrike();
   assert true g7.frame->at(3).isStrike();
   assert equals g7.frame->at(2).score() 24;
   assert equals g7.frame->at(3).score() 34;
   assert equals g7.score() 34;

   t5 := new NewThrow(knockedPins:=5, game:=g7);
   assert occurrence t5;
   assert equals g7.frame->at(1).score() 4;
   assert equals g7.frame->at(2).score() 29;
   assert equals g7.frame->at(3).score() 44;
   assert equals g7.frame->at(4).score() 49;
   assert equals g7.score() 49;


   t6 := new NewThrow(knockedPins:=3, game:=g7);
   assert occurrence t6;
   assert equals g7.frame->at(1).score() 4;
   assert equals g7.frame->at(2).score() 29;
   assert equals g7.frame->at(3).score() 47;
   assert equals g7.frame->at(4).score() 55;
   assert equals g7.score() 55;

   assert true g7.finished()=false;

 }
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)
- IncompleteGameWithRegularAndNonTerminalSpareFrames (S3)
- IncompleteGameWithTwoConsecutiveAndNonTerminalSpares (S4)
- IncompleteGameWithStrikeInFirstFrame (S5)
- IncompleteGameWithRegularAndNonTerminalStrikeFrames (S6)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- ● **Current Test Case. Line 37**: The result is 24 but it is expected to be 29 in assertion *assert equals g7.frame->at(2).score() 29;*

▶ The conceptual modeler modifies the derivation rule of the attribute *Frame::score* because it does not allow two consecutive strikes.

```
class Frame
…
score():Integer=
                let knockedPins:Integer=
                       self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                        if self.isSpare() then
                           self.next().throw->sortedBy(order)->at(1).knockedPins
                           else 0
                        endif
                       else 0
                       endif
                in

                    let strikeBonus=
                        if self.throw->notEmpty() and self.isStrike() then
                            if self.throw->first().next().isDefined() then
                                self.throw->first().next().knockedPins
                                +
                                if self.throw->first().next().next().isDefined() then
                                    self.throw->first().next().next().knockedPins
                                else 0
                                endif
                            else 0
                            endif
                       else
                        0
                       endif
                in
                    knockedPins+spareBonus+strikeBonus+scoreAtPreviousFrame


…
end
```

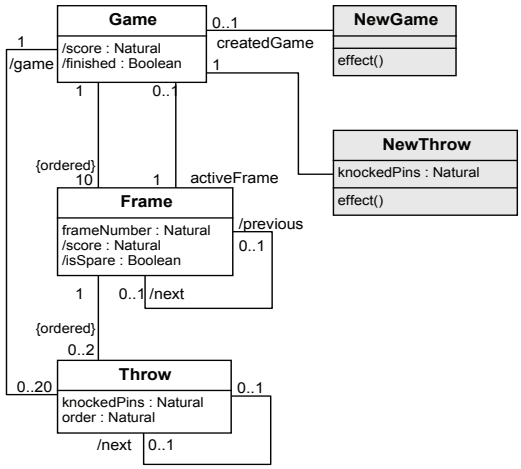
- **CSUT**: Undefined property *next* in expression {Throw}.next

▶ The conceptual modeler adds the knowledge about the next throw of a throw.

```
class Throw
attributes
knockedPins:Integer
 order:Integer
operations
game():Game=self.frame.game
next():Throw=if self.order=2 or self.frame.isStrike() then
                self.frame.next().throw->sortedBy(order)->at(1)
             else
                self.frame.throw->sortedBy(order)->at(2)
             endif
end
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema




```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
                then 0
                else self.activeFrame.score
                endif

context Game::finished:Boolean
  derive: self.frame.throw->size()=20

context Frame::isSpare:Boolean
  derive: self.throw->size()>1
          and self.throw->at(1).knockedPins  + self.throw->at(2).knockedPins=10

context Frame::isStrike:Boolean
  derive: self.throw->size()=1
                and self.throw->sortedBy(order)->at(1).knockedPins=10
context Frame::score:Integer
  derive:       let knockedPins:Integer=
                        self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                        if self.isSpare() then
                           self.next().throw->sortedBy(order)->at(1).knockedPins
```

```
                        else 0
                        endif
                       else 0
                       endif
                in
                let strikeBonus=
                        if self.throw->notEmpty() and self.isStrike() then
                            if self.throw->first().next().isDefined() then
                                self.throw->first().next().knockedPins
                                +
                                if self.throw->first().next().next().isDefined() then
                                    self.throw->first().next().next().knockedPins
                                else 0
                                endif
                            else 0
                            endif
                       else
                        0
                       endif
                in
                    knockedPins+spareBonus+strikeBonus+scoreAtPreviousFrame

context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)

context Frame::next:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber+1)

context Throw::next:Throw
  derive: if self.order=2 or self.frame.isStrike() then
              self.frame.next().throw->sortedBy(order)->at(1)
          else
              self.frame.throw->sortedBy(order)->at(2)
          endif


context Throw::game:Game
  derive: self.frame.game

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
             and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
              and t.game()=self.game
              and t.knockedPins=self.knockedPins
              and t.frame=t.game().frame->sortedBy(frameNumber)
                        ->reject(f|f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10)
                        ->first()
              and t.order=t.frame.throw->size()
     and t.game().activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	11MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD	A derived type involved in a test case does not exist in the CSUD	An event type involved in a test case does not exist in the CSUD
The basic type is relevant and it is added to the CSUD	The derived type is relevant and it is added to the CSUD	The event type is relevant and it is added to the CSUD
	1	

Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
An assertion about the IB state fails or contains an error			**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect		A precondition is added/updated	The expression is corrected	The CSUD is changed
	1				

Iteration 9

Input conceptual schema

The resultant conceptual schema of Iteration 8.

Iteration objective

S8: A complete game with regular, non-terminal strike and non-terminal spare frames

Current Test case

```
testprogram CompleteGameWithRegularAndNonTerminalStrikesAndSpares{

test S8{
   ng8 := new NewGame;
   assert occurrence ng8;
   g8 := ng8.createdGame;

   t1 := new NewThrow(knockedPins:=8, game:=g8);
   assert occurrence t1;
   assert equals g8.frame->at(1).score() 8;
   assert equals g8.score() 8;

   t2 := new NewThrow(knockedPins:=0, game:=g8);
   assert occurrence t2;
   assert equals g8.frame->at(1).score() 8;
   assert equals g8.score() 8;

   t3 := new NewThrow(knockedPins:=3, game:=g8);
   assert occurrence t3;
   assert equals g8.frame->at(1).score() 8;
   assert equals g8.frame->at(2).score() 11;
   assert equals g8.score() 11;

   t4 := new NewThrow(knockedPins:=7, game:=g8);
   assert occurrence t4;
   assert true g8.frame->at(2).isSpare();
   assert equals g8.frame->at(1).score() 8;
   assert equals g8.frame->at(2).score() 18;
   assert equals g8.score() 18;
```

t5 := new NewThrow(knockedPins:=10, game:=g8);
assert occurrence t5;
assert true g8.frame->at(3).isStrike();
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 38;
assert equals g8.score() 38;

t6 := new NewThrow(knockedPins:=3, game:=g8);
assert occurrence t6;
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 41;
assert equals g8.frame->at(4).score() 44;
assert equals g8.score() 44;

t7 := new NewThrow(knockedPins:=7, game:=g8);
assert occurrence t7;
assert true g8.frame->at(4).isSpare();
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 58;
assert equals g8.score() 58;

t8 := new NewThrow(knockedPins:=10, game:=g8);
assert occurrence t8;
assert true g8.frame->at(5).isStrike();
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 68;
assert equals g8.frame->at(5).score() 78;
assert equals g8.score() 78;

t9 := new NewThrow(knockedPins:=5, game:=g8);
assert occurrence t9;
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 68;
assert equals g8.frame->at(5).score() 83;
assert equals g8.frame->at(6).score() 88;
assert equals g8.score() 88;

t10 := new NewThrow(knockedPins:=2, game:=g8);
assert occurrence t10;
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 68;
assert equals g8.frame->at(5).score() 85;
assert equals g8.frame->at(6).score() 92;
assert equals g8.score() 92;

t11 := new NewThrow(knockedPins:=7, game:=g8);
assert occurrence t11;
assert equals g8.frame->at(1).score() 8;

```
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 68;
assert equals g8.frame->at(5).score() 85;
assert equals g8.frame->at(6).score() 92;
assert equals g8.frame->at(7).score() 99;
assert equals g8.score() 99;

t12 := new NewThrow(knockedPins:=1, game:=g8);
assert occurrence t12;
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 68;
assert equals g8.frame->at(5).score() 85;
assert equals g8.frame->at(6).score() 92;
assert equals g8.frame->at(7).score() 100;
assert equals g8.score() 100;

t13 := new NewThrow(knockedPins:=0, game:=g8);
assert occurrence t13;
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 68;
assert equals g8.frame->at(5).score() 85;
assert equals g8.frame->at(6).score() 92;
assert equals g8.frame->at(7).score() 100;
assert equals g8.frame->at(8).score() 100;
assert equals g8.score() 100;

t14 := new NewThrow(knockedPins:=10, game:=g8);
assert occurrence t14;
assert true g8.frame->at(8).isSpare();
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 68;
assert equals g8.frame->at(5).score() 85;
assert equals g8.frame->at(6).score() 92;
assert equals g8.frame->at(7).score() 100;
assert equals g8.frame->at(8).score() 110;
assert equals g8.score() 110;

t15 := new NewThrow(knockedPins:=10, game:=g8);
assert occurrence t15;
assert true g8.frame->at(9).isStrike();
assert equals g8.frame->at(1).score() 8;
assert equals g8.frame->at(2).score() 28;
assert equals g8.frame->at(3).score() 48;
assert equals g8.frame->at(4).score() 68;
assert equals g8.frame->at(5).score() 85;
assert equals g8.frame->at(6).score() 92;
assert equals g8.frame->at(7).score() 100;
assert equals g8.frame->at(8).score() 120;
assert equals g8.frame->at(9).score() 130;
assert equals g8.score() 130;
```

```
    t16 := new NewThrow(knockedPins:=2, game:=g8);
    assert occurrence t16;
    assert equals g8.frame->at(1).score() 8;
    assert equals g8.frame->at(2).score() 28;
    assert equals g8.frame->at(3).score() 48;
    assert equals g8.frame->at(4).score() 68;
    assert equals g8.frame->at(5).score() 85;
    assert equals g8.frame->at(6).score() 92;
    assert equals g8.frame->at(7).score() 100;
    assert equals g8.frame->at(8).score() 120;
    assert equals g8.frame->at(9).score() 132;
    assert equals g8.frame->at(10).score() 134;
    assert equals g8.score() 134;

    t17 := new NewThrow(knockedPins:=3, game:=g8);
    assert occurrence t17;
    assert equals g8.frame->at(1).score() 8;
    assert equals g8.frame->at(2).score() 28;
    assert equals g8.frame->at(3).score() 48;
    assert equals g8.frame->at(4).score() 68;
    assert equals g8.frame->at(5).score() 85;
    assert equals g8.frame->at(6).score() 92;
    assert equals g8.frame->at(7).score() 100;
    assert equals g8.frame->at(8).score() 120;
    assert equals g8.frame->at(9).score() 135;
    assert equals g8.frame->at(10).score() 140;
    assert equals g8.score() 140;

    assert true g8.finished()=true;
  }
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)
- IncompleteGameWithRegularAndNonTerminalSpareFrames (S3)
- IncompleteGameWithTwoConsecutiveAndNonTerminalSpares (S4)
- IncompleteGameWithStrikeInFirstFrame (S5)
- IncompleteGameWithRegularAndNonTerminalStrikeFrames (S6)
- IncompleteGameWithTwoConsecutiveAndNonTerminalStrikes (S7)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

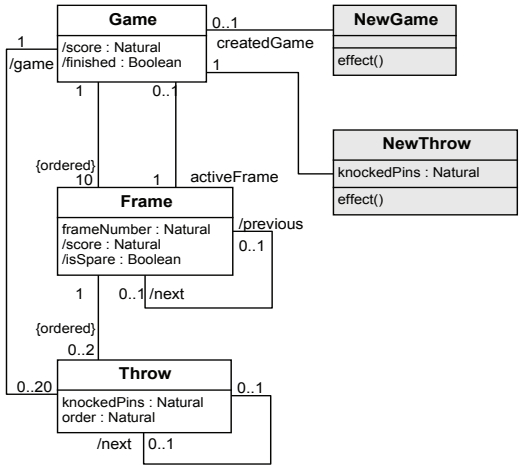
- **Current Test Case. Line 176**: Assert expression is false but it is expected to be true in expression *assert true g8.finished()=true;*

- The conceptual modeler realizes that a game is not only finished when there are 20 throws. The derived attribute *Game::finished* is changed to include the correct knowledge taking into account that there may be strike frames.

```
class Game
…
finished():Boolean=self.frame->at(10).throw->size()=2
end
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema




```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
               then 0
               else self.activeFrame.score
               endif

context Game::finished:Boolean
  derive: self.frame->at(10).throw->size()=2

context Frame::isSpare:Boolean
  derive: self.throw->size()>1
          and self.throw->at(1).knockedPins  + self.throw->at(2).knockedPins=10

context Frame::isStrike:Boolean
  derive: self.throw->size()=1
                    and self.throw->sortedBy(order)->at(1).knockedPins=10

context Frame::score:Integer
  derive:       let knockedPins:Integer=
                        self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                        if self.isSpare() then
                           self.next().throw->sortedBy(order)->at(1).knockedPins
                        else 0
                        endif
                       else 0
                       endif
                in
                let strikeBonus=
```

```
                        if self.throw->notEmpty() and self.isStrike() then
                            if self.throw->first().next().isDefined() then
                                self.throw->first().next().knockedPins
                                +
                                if self.throw->first().next().next().isDefined() then
                                    self.throw->first().next().next().knockedPins
                                else 0
                                endif
                            else 0
                            endif
                        else
                         0
                        endif
                in
                    knockedPins+spareBonus+strikeBonus+scoreAtPreviousFrame

context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)

context Frame::next:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber+1)

context Throw::next:Throw
  derive: if self.order=2 or self.frame.isStrike() then
              self.frame.next().throw->sortedBy(order)->at(1)
          else
              self.frame.throw->sortedBy(order)->at(2)
          endif


context Throw::game:Game
  derive: self.frame.game

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
          and self.createdGame=g
          and g.frame->size()=10
          and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
          and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->sortedBy(frameNumber)
                       ->reject(f|f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10)
                       ->first()
             and t.order=t.frame.throw->size()
     and t.game().activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	8MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD		**A derived type involved in a test case does not exist in the CSUD**		**An event type involved in a test case does not exist in the CSUD**	
The basic type is relevant and it is added to the CSUD		The derived type is relevant and it is added to the CSUD		The event type is relevant and it is added to the CSUD	
Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
An assertion about the IB state fails or contains an error			**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect		A precondition is added/updated	The expression is corrected	The CSUD is changed
	1				

Iteration 10

Input conceptual schema

The resultant conceptual schema of Iteration 9.

Iteration objective

S9: A complete game with a spare in the last frame

Current Test case

```
testprogram CompleteGameWithSpareInLastFrame{

test S9{
   ng9 := new NewGame;
   assert occurrence ng9;
   g9 := ng9.createdGame;

   t1 := new NewThrow(knockedPins:=4, game:=g9);
   assert occurrence t1;
   assert equals g9.frame->at(1).score() 4;
   assert equals g9.score() 4;

   t2 := new NewThrow(knockedPins:=3, game:=g9);
   assert occurrence t2;
   assert equals g9.frame->at(1).score() 7;
   assert equals g9.score() 7;

   t3 := new NewThrow(knockedPins:=0, game:=g9);
   assert occurrence t3;
   assert equals g9.frame->at(1).score() 7;
   assert equals g9.frame->at(2).score() 7;
   assert equals g9.score() 7;
```

```
    t4 := new NewThrow(knockedPins:=2, game:=g9);
    assert occurrence t4;
    assert equals g9.frame->at(1).score() 7;
    assert equals g9.frame->at(2).score() 9;
    assert equals g9.score() 9;

     index:=0;
     expectedGameScore:=9;
     while index<14 do
       tx:=new NewThrow(knockedPins:=3, game:=g9);
       assert occurrence tx;
       index:=index+1;
       expectedGameScore:=expectedGameScore+3;
       assert equals g9.score() expectedGameScore;
     endwhile

    t19 := new NewThrow(knockedPins:=3, game:=g9);
    assert occurrence t19;
    assert equals g9.frame->at(10).score() 54;
    assert equals g9.score() 54;

    t20 := new NewThrow(knockedPins:=7, game:=g9);
    assert occurrence t20;
    assert equals g9.frame->at(10).score() 61;
    assert equals g9.score() 61;

    assert true g9.finished()=false;

    t21 := new NewThrow(knockedPins:=4, game:=g9);
    assert occurrence t21;
    assert equals g9.frame->at(10).score() 65;
    assert equals g9.score() 65;

    assert true g9.finished()=true;
    }
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)
- IncompleteGameWithRegularAndNonTerminalSpareFrames (S3)
- IncompleteGameWithTwoConsecutiveAndNonTerminalSpares (S4)
- IncompleteGameWithStrikeInFirstFrame (S5)
- IncompleteGameWithRegularAndNonTerminalStrikeFrames (S6)
- IncompleteGameWithTwoConsecutiveAndNonTerminalStrikes (S7)
- CompleteGameWithRegularAndNonTerminalStrikesAndSpares (S8)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- **Methods**: Expression *t.frame.game.activeFrame* before '.' operation (t.frame) should be an object expression.

▶ The conceptual modeler realizes that additional throws (allowed in spare terminal frames) have not assigned its frame correctly. Given that the frame of a throw is the last non-finished frame, the modeler changes the derivation rule of the attribute *Frame::finished*, because it does not take into account this particular situation.

```
class Frame
…
finished():Boolean=if (self.isLastFrame() and (self.isStrike() or self.isSpare())) then
                      self.throw->size()=3
                   else self.throw->size=2 or self.throw.knockedPins->sum()=10
                   endif

end
```

- **CSUT**: Undefined property *isLastFrame* in expression *Frame::isLastFrame*

▶ The conceptual modeler specifies the derived attribute *Frame::isLastFrame*

```
class Frame
…
isLastFrame():Boolean=(frameNumber=10)
…
End
```

- **CSUT**: The postcondition of *NewThrow* is false.

▶ The conceptual modeler modifies the postcondition according to the new definition of the attribute *Frame::finished()*

```
context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->sortedBy(frameNumber)->reject(f|
                           if (f.isLastFrame() and
                              (f.throw@pre->size()=1
                               and f.throw@pre->sortedBy(order)->at(1).knockedPins=10
                               or f.throw@pre->size()>1 and f.throw@pre->at(1).knockedPins
                               + f.throw@pre->at(2).knockedPins=10))
                           then f.throw@pre->size()=3
                           else f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10
                           endif)->first()
             and t.order=t.frame.throw->size()
             and t.game().activeFrame=t.frame)
```

- **Current test case. Line 53**: Inconsistent state after t21:NewThrow event execution: Multiplicity constraint violation in association `frame_throw': Object `oid4' of class `Frame' is connected to 3 objects of class `Throw' but the multiplicity is specified as `0..2'.

▶ The conceptual modeler realizes that, in this particular case, a frame may have three throws. The schema is changed to allow this situation.

```
association frame_throw between
  Frame[1]
  Throw[0..3] ordered
end

context Frame inv nonLastFramesHaveAtMostTwoThrows:
(not self.isLastFrame()) implies self.throw->size()<=2
```

- **Current Test Case. Line 50**: Assert expression is false but is expected to be true in *assert true g9.finished()=false*;

▶ The conceptual modeler modifies the derivation rule of the attribute *Game::finished*, taking into account that the last frame may have three throws.

```
class Game
…
finished():Boolean=self.frame->at(10).finished()
end
```

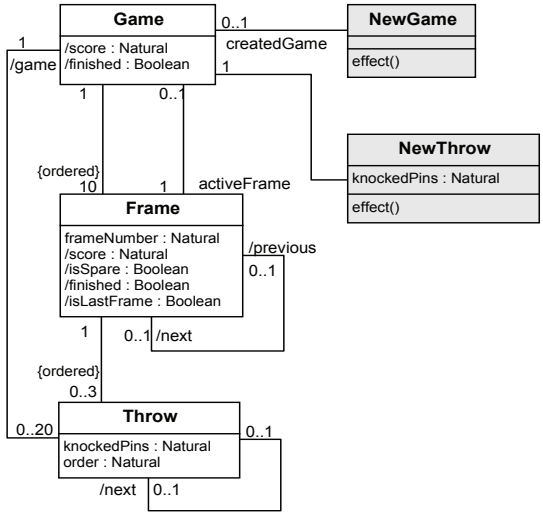
- **Current Test Case. Line 57**: Assert expression is false but is expected to be true in *assert true g9.finished()=true*;

▶ The conceptual modeler realizes that a spare is not correctly recognized because the ascendant creation order of throws is not explicitly specified in the schema.

```
class Frame
…
isSpare():Boolean=self.throw->size()>1
                  and self.throw->sortedBy(order)->at(1).knockedPins
                      + self.throw->sortedBy(order)->at(2).knockedPins=10
…
end
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema



```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
               then 0
               else self.activeFrame.score
               endif

context Game::finished:Boolean
  derive: self.frame->at(10).finished()

context Frame::isSpare:Boolean
  derive: self.throw->size()>1
          and self.throw->sortedBy(order)->at(1).knockedPins
               + self.throw->sortedBy(order)->at(2).knockedPins=10

context Frame::isStrike:Boolean
  derive: self.throw->size()=1
                    and self.throw->sortedBy(order)->at(1).knockedPins=10

context Frame::score:Integer
  derive:       let knockedPins:Integer=
                        self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                        if self.isSpare() then
                           self.next().throw->sortedBy(order)->at(1).knockedPins
                        else 0
                        endif
                       else 0
                       endif
                in
                let strikeBonus=
                        if self.throw->notEmpty() and self.isStrike() then
                            if self.throw->first().next().isDefined() then
                                self.throw->first().next().knockedPins
                                +
                                if self.throw->first().next().next().isDefined() then
                                    self.throw->first().next().next().knockedPins
                                else 0
                                endif
                            else 0
                            endif
                       else
                        0
                       endif
                in
                    knockedPins+spareBonus+strikeBonus+scoreAtPreviousFrame


context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)

context Frame::next:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber+1)


context Frame::finished:Boolean
  derive: if (self.isLastFrame() and (self.isStrike() or self.isSpare())) then
                     self.throw->size()=3
                  else self.throw->size=2 or self.throw.knockedPins->sum()=10
                  endif

context Frame::isLastFrame:Boolean
  derive: (frameNumber=10)

context Frame inv nonLastFramesHaveAtMostTwoThrows:
(not self.isLastFrame()) implies self.throw->size()<=2


context Throw::next:Throw
  derive: if self.order=2 or self.frame.isStrike() then
               self.frame.next().throw->sortedBy(order)->at(1)
            else
               self.frame.throw->sortedBy(order)->at(2)
            endif
```

```
context Throw::game:Game
  derive: self.frame.game

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
             and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
              and t.game()=self.game
              and t.knockedPins=self.knockedPins
              and t.frame=t.game().frame->sortedBy(frameNumber)->reject(f|
                            if (f.isLastFrame() and
                               (f.throw@pre->size()=1
                                and f.throw@pre->sortedBy(order)->at(1).knockedPins=10
                                or f.throw@pre->size()>1 and f.throw@pre->at(1).knockedPins
                                + f.throw@pre->at(2).knockedPins=10))
                            then f.throw@pre->size()=3
                            else f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10
                            endif)->first()
              and t.order=t.frame.throw->size()
              and t.game().activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	17MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD		**A derived type involved in a test case does not exist in the CSUD**		**An event type involved in a test case does not exist in the CSUD**	
The basic type is relevant and it is added to the CSUD		The derived type is relevant and it is added to the CSUD		The event type is relevant and it is added to the CSUD	
		1			
Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
			1		**1**
An assertion about the IB state fails or contains an error			**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect		A precondition is added/updated	The expression is corrected	The CSUD is changed
	1				**1**

Iteration 11

Input conceptual schema

The resultant conceptual schema of Iteration 10.

Iteration objective

S10: A complete game with a strike in the last frame

Current Test case

```
testprogram CompleteGameWithStrikeInLastFrame{

test S10{
   ng10 := new NewGame;
   assert occurrence ng10;
   g10 := ng10.createdGame;

   t1 := new NewThrow(knockedPins:=4, game:=g10);
   assert occurrence t1;
   assert equals g10.frame->at(1).score() 4;
   assert equals g10.score() 4;

   t2 := new NewThrow(knockedPins:=3, game:=g10);
   assert occurrence t2;
   assert equals g10.frame->at(1).score() 7;
   assert equals g10.score() 7;

   t3 := new NewThrow(knockedPins:=0, game:=g10);
   assert occurrence t3;
   assert equals g10.frame->at(1).score() 7;
   assert equals g10.frame->at(2).score() 7;
   assert equals g10.score() 7;

   t4 := new NewThrow(knockedPins:=2, game:=g10);
   assert occurrence t4;
   assert equals g10.frame->at(1).score() 7;
   assert equals g10.frame->at(2).score() 9;
   assert equals g10.score() 9;

    index:=0;
    expectedGameScore:=9;
    while index<14 do
      tx:=new NewThrow(knockedPins:=3, game:=g10);
      assert occurrence tx;
      index:=index+1;
      expectedGameScore:=expectedGameScore+3;
      assert equals g10.score() expectedGameScore;
    endwhile

   assert true g10.finished()=false;
```

```
    t19 := new NewThrow(knockedPins:=10, game:=g10);
    assert occurrence t19;
    assert equals g10.frame->at(10).score() 61;
    assert equals g10.score() 61;

    assert true g10.finished()=false;

    t20 := new NewThrow(knockedPins:=2, game:=g10);
    assert occurrence t20;
    assert equals g10.frame->at(10).score() 63;
    assert equals g10.score() 63;
    assert true g10.frame->at(10).isStrike();
    assert true g10.finished()=false;

    t21 := new NewThrow(knockedPins:=4, game:=g10);
    assert occurrence t21;
    assert equals g10.frame->at(10).score() 67;
    assert equals g10.score() 67;

    assert true g10.finished()=true;
  }
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)
- IncompleteGameWithRegularAndNonTerminalSpareFrames (S3)
- IncompleteGameWithTwoConsecutiveAndNonTerminalSpares (S4)
- IncompleteGameWithStrikeInFirstFrame (S5)
- IncompleteGameWithRegularAndNonTerminalStrikeFrames (S6)
- IncompleteGameWithTwoConsecutiveAndNonTerminalStrikes (S7)
- CompleteGameWithRegularAndNonTerminalStrikesAndSpares (S8)
- CompleteGameWithSpareInLastFrame (S9)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

- **Methods**: Expression *t.frame.game.activeFrame* before '.' operation (t.frame) should be an object expression.

- The conceptual modeler realizes that additional throws (allowed in strike terminal frames) have not assigned its frame correctly because an strike is not correctly identified in the last frame of a game. Then, the modeler changes the derivation rule *Frame::isStrike*

```
class Frame
…
isStrike():Boolean=if self.isLastFrame() then
                        self.throw->sortedBy(order)->at(1).knockedPins=10
                   else self.throw->size()=1
                        and self.throw->sortedBy(order)->at(1).knockedPins=10
                   endif
end
```

- **CSUT**: The postcondition of *NewThrow* is false.

- The conceptual modeler modifies the postcondition according to the new definition of the attribute *Frame::isStrike*

```
context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->sortedBy(frameNumber)->reject(f|
                           if (f.isLastFrame() and
                              ((if f.isLastFrame() then
                                  f.throw@pre->sortedBy(order)->at(1).knockedPins=10
                                else f.throw@pre->size()=1
                                     and f.throw@pre->sortedBy(order)->at(1).knockedPins=10
                                endif)
                                or
                                (f.throw@pre->size()>1
                                and f.throw@pre->sortedBy(order)->at(1).knockedPins
                                     + f.throw@pre->sortedBy(order)->at(2).knockedPins=10
                                )
                                )) then
                                  f.throw@pre->size()=3
                           else f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10
                           endif)->first()
             and t.order=t.frame.throw->size()
             and t.game().activeFrame=t.frame)
```

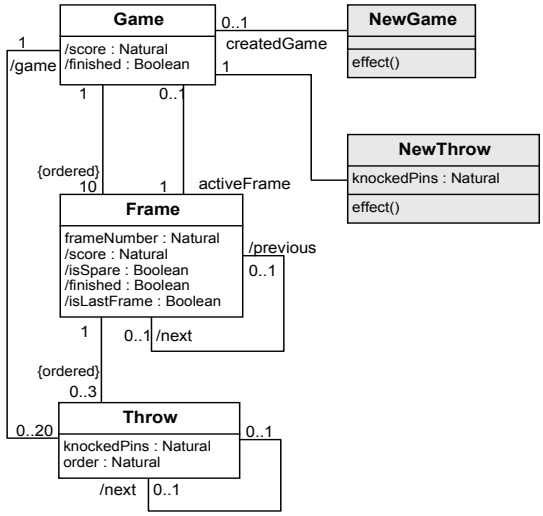
- The verdict of the current test set is *Pass*.

- The conceptual modeler refactors the derivation rule *Frame::isStrike* giving that, in fact, a strike may be identified only by checking that in the first throw of a frame, the number of knocked pins is 10.

```
class Frame
…
isStrike():Boolean=self.throw->sortedBy(order)->at(1).knockedPins=10
end
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema




```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
                then 0
                else self.activeFrame.score
                endif

context Game::finished:Boolean
  derive: self.frame->at(10).finished()

context Frame::isSpare:Boolean
  derive: self.throw->size()>1
          and self.throw->sortedBy(order)->at(1).knockedPins
                + self.throw->sortedBy(order)->at(2).knockedPins=10

context Frame::isStrike:Boolean
  derive: self.throw->sortedBy(order)->at(1).knockedPins=10

context Frame::score:Integer
  derive:       let knockedPins:Integer=
                        self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                        if self.isSpare() then
                           self.next().throw->sortedBy(order)->at(1).knockedPins
                        else 0
                        endif
                       else 0
                       endif
                in
                let strikeBonus=
                        if self.throw->notEmpty() and self.isStrike() then
                            if self.throw->first().next().isDefined() then
                                self.throw->first().next().knockedPins
                                +
                                if self.throw->first().next().next().isDefined() then
                                    self.throw->first().next().next().knockedPins
                                else 0
                                endif
                            else 0
                            endif
                       else
                        0
                       endif
                in
                    knockedPins+spareBonus+strikeBonus+scoreAtPreviousFrame
```

```
context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)

context Frame::next:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber+1)


context Frame::finished:Boolean
  derive: if (self.isLastFrame() and (self.isStrike() or self.isSpare())) then
                     self.throw->size()=3
                  else self.throw->size=2 or self.throw.knockedPins->sum()=10
                  endif

context Frame::isLastFrame:Boolean
  derive: (frameNumber=10)

context Frame inv nonLastFramesHaveAtMostTwoThrows:
(not self.isLastFrame()) implies self.throw->size()<=2


context Throw::next:Throw
  derive: if self.order=2 or self.frame.isStrike() then
               self.frame.next().throw->sortedBy(order)->at(1)
            else
               self.frame.throw->sortedBy(order)->at(2)
            endif


context Throw::game:Game
  derive: self.frame.game

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
            and self.createdGame=g
            and g.frame->size()=10
            and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
            and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->sortedBy(frameNumber)->reject(f|
                           if (f.isLastFrame() and
                              ((if f.isLastFrame() then
                                  f.throw@pre->sortedBy(order)->at(1).knockedPins=10
                               else f.throw@pre->size()=1
                                    and f.throw@pre->sortedBy(order)->at(1).knockedPins=10
                               endif)
                               or
                               (f.throw@pre->size()>1
                               and f.throw@pre->sortedBy(order)->at(1).knockedPins
                                    + f.throw@pre->sortedBy(order)->at(2).knockedPins=10
                               )
                               )) then
                                  f.throw@pre->size()=3
                           else f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10
                           endif)->first()
             and t.order=t.frame.throw->size()
             and t.game().activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	9MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD		**A derived type involved in a test case does not exist in the CSUD**		**An event type involved in a test case does not exist in the CSUD**	
The basic type is relevant and it is added to the CSUD		The derived type is relevant and it is added to the CSUD		The event type is relevant and it is added to the CSUD	
Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
					1
An assertion about the IB state fails or contains an error			**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect		A precondition is added/updated	The expression is corrected	The CSUD is changed
	1				

Iteration 12

Input conceptual schema

The resultant conceptual schema of Iteration 11.

Iteration objective

S11: Throws are not allowed for finished games

Current Test case

```
testprogram CompleteGameWithoutSparesAndStrikes{

 test S1{
   ...
    assert true g1.finished()=true;

    //Extension
    t21 := new NewThrow(knockedPins:=4, game:=g1);
    assert non-occurrence t21;
 }
}
```

Regression test cases

- CompleteGameWithoutSparesAndStrikes (S1)
- IncompleteGameWithASpareInTheFirstFrame (S2)

- IncompleteGameWithRegularAndNonTerminalSpareFrames (S3)
- IncompleteGameWithTwoConsecutiveAndNonTerminalSpares (S4)
- IncompleteGameWithStrikeInFirstFrame (S5)
- IncompleteGameWithRegularAndNonTerminalStrikeFrames (S6)
- IncompleteGameWithTwoConsecutiveAndNonTerminalStrikes (S7)
- CompleteGameWithRegularAndNonTerminalStrikesAndSpares (S8)
- CompleteGameWithSpareInLastFrame (S9)
- CompleteGameWithStrikeInLastFrame (S10)

Test-Driven evolution of the CSUD

● Error ● Fail ● Pass ▶Schema change

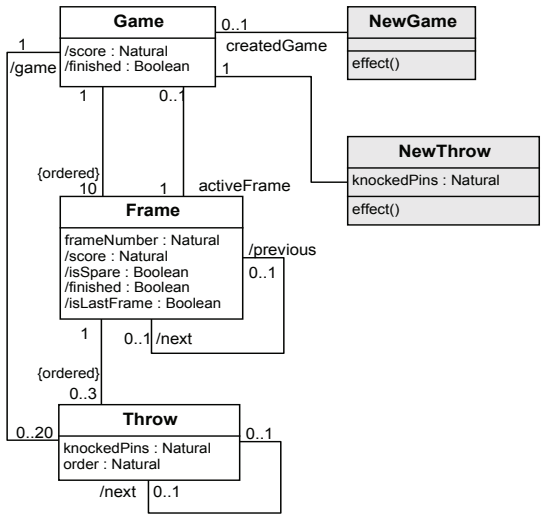
- **Current Test Case. Line 47**: Preconditions of the domain event t21:NewThrow are satisfied and consequently, the event can occur.

▶ The conceptual modeler realizes that an initial integrity constraint is needed to prevent the occurrence of the event NewThrow when the game is finished. …

```
context NewThrow ini inv isNotAllowedIfTheGameIsFinished:
        not (self.game.finished())
```

- The verdict of the current test set is *Pass*.

Resultant conceptual schema



```
context Game::score:Integer
  derive: if self.activeFrame->isEmpty()
               then 0
               else self.activeFrame.score
               endif

context Game::finished:Boolean
  derive: self.frame->at(10).finished()

context Frame::isSpare:Boolean
  derive: self.throw->size()>1
          and self.throw->sortedBy(order)->at(1).knockedPins
               + self.throw->sortedBy(order)->at(2).knockedPins=10

context Frame::isStrike:Boolean
  derive: self.throw->sortedBy(order)->at(1).knockedPins=10

context Frame::score:Integer
  derive:       let knockedPins:Integer=
                       self.throw.knockedPins->sum()
                in
                let scoreAtPreviousFrame=
                       if self.previous().isUndefined() then 0
                       else
                        self.previous().score()
                       endif
                in
                let spareBonus=
                       if self.next().throw->sortedBy(order)->at(1).isDefined() then
                        if self.isSpare() then
                           self.next().throw->sortedBy(order)->at(1).knockedPins
                        else 0
                        endif
                       else 0
                       endif
                in
                let strikeBonus=
                        if self.throw->notEmpty() and self.isStrike() then
                            if self.throw->first().next().isDefined() then
                                self.throw->first().next().knockedPins
                                +
                                if self.throw->first().next().next().isDefined() then
                                    self.throw->first().next().next().knockedPins
                                else 0
                                endif
                            else 0
                            endif
                       else
                        0
                       endif
                in
                    knockedPins+spareBonus+strikeBonus+scoreAtPreviousFrame


context Frame::previous:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber-1)

context Frame::next:Frame
  derive: self.game.frame->any(f|f.frameNumber=self.frameNumber+1)


context Frame::finished:Boolean
  derive: if (self.isLastFrame() and (self.isStrike() or self.isSpare())) then
                     self.throw->size()=3
                  else self.throw->size=2 or self.throw.knockedPins->sum()=10
                  endif

context Frame::isLastFrame:Boolean
  derive: (frameNumber=10)

context Frame inv nonLastFramesHaveAtMostTwoThrows:
(not self.isLastFrame()) implies self.throw->size()<=2


context Throw::next:Throw
  derive: if self.order=2 or self.frame.isStrike() then
               self.frame.next().throw->sortedBy(order)->at(1)
            else
               self.frame.throw->sortedBy(order)->at(2)
            endif
```

```
context Throw::game:Game
  derive: self.frame.game

context NewGame::effect()
post:
(Game.allInstances- Game.allInstances@pre)
   ->one(g | g.oclIsNew()
             and self.createdGame=g
             and g.frame->size()=10
             and g.frame->sortedBy(frameNumber).frameNumber=Sequence{1,2,3,4,5,6,7,8,9,10}
             and g.activeFrame=g.frame->sortedBy(frameNumber)->first())

context NewThrow ini inv isNotAllowedIfTheGameIsFinished:
        not (self.game.finished())


context NewThrow::effect()
post:
(Throw.allInstances- Throw.allInstances@pre)
   ->one(t | t.oclIsNew()
             and t.game()=self.game
             and t.knockedPins=self.knockedPins
             and t.frame=t.game().frame->sortedBy(frameNumber)->reject(f|
                           if (f.isLastFrame() and
                              ((if f.isLastFrame() then
                                  f.throw@pre->sortedBy(order)->at(1).knockedPins=10
                               else f.throw@pre->size()=1
                                    and f.throw@pre->sortedBy(order)->at(1).knockedPins=10
                               endif)
                               or
                               (f.throw@pre->size()>1
                               and f.throw@pre->sortedBy(order)->at(1).knockedPins
                                    + f.throw@pre->sortedBy(order)->at(2).knockedPins=10
                               )
                               )) then
                                  f.throw@pre->size()=3
                           else f.throw@pre->size=2 or f.throw@pre.knockedPins->sum()=10
                           endif)->first()
             and t.order=t.frame.throw->size()
             and t.game().activeFrame=t.frame)
```

Additional information

TIME TO FINISH THE ITERATION	6MIN

ERRORS AND FAILURES THAT DRIVE THE CONCEPTUAL MODELING ACTIVITY

A basic type involved in a test case does not exist in the CSUD		**A derived type involved in a test case does not exist in the CSUD**		**An event type involved in a test case does not exist in the CSUD**	
The basic type is relevant and it is added to the CSUD		The derived type is relevant and it is added to the CSUD		The event type is relevant and it is added to the CSUD	
Inconsistent state before the occurrence of an event		**Inconsistent state after the occurrence of an event**		**The postcondition of an event is not satisfied.**	
Some static constraint is invalid and it is modified.	Some initial integrity constraint is invalid and it is modified.	The event postcondition/method is incorrect and it is modified.	Some constraint is invalid and it is modified.	The method is not correct and it is modified.	The postcondition is not correct and it is modified.
An assertion about the IB state fails or contains an error			**Assert non-occurrence fails**	**Semantic error in an expression**	
The effect of an event type is not correct	A derivation rule is incorrect		A precondition is added/updated	The expression is corrected	The CSUD is changed
			1		

4. Experimentation analysis

In this section, we analyze the properties of the resultant conceptual schema, the testing effort, the kinds of errors and failures and the characteristics of the TDCM iterations, which has been performed in order to achieve the conceptual schema of the bowling game case study.

4.1. The resultant conceptual schema

The resultant conceptual schema of the TDCM application is the schema obtained in the last iteration (see the subsection "Resultant schema" of the iteration 12 in Section 3). You can download a zipped file with the CSTLProcessor and the case study files (the resultant conceptual schema in the USEx executable format, the methods file, and the CSTL test programs).

We finished the TDCM application when two conditions hold: 1) we formalized as test cases all the representative stories according to our testing strategy, 2) the verdict of all the test cases became *Pass*. In order to reach the verdicts report of Figure 3, we performed **12 iterations** that we analyze in the following section. **1,78 hours** have been invested in specifying the test cases in the CSUT language and **4,59 hours** have been invested in the development of the TDCM iterations.

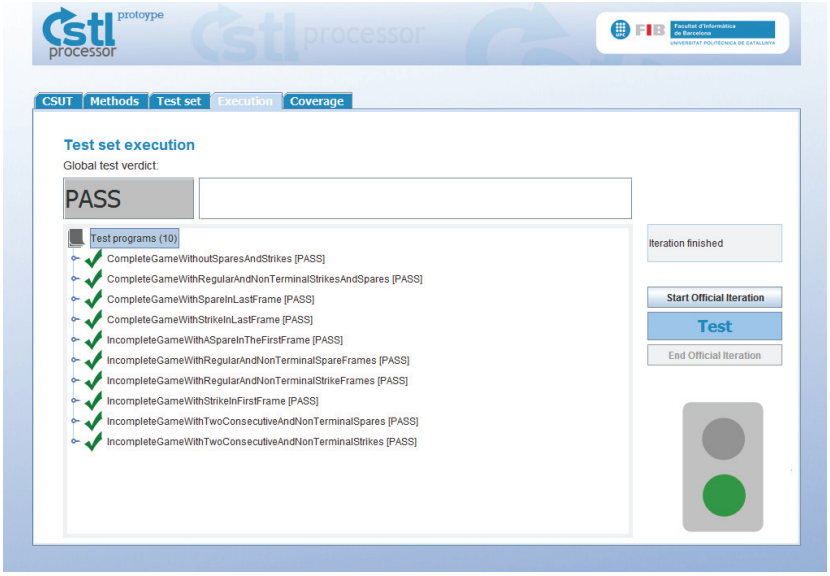


Fig. 3. Test case execution report provided by the CSTL processor at the end of the last TDCM iteration

Quality properties of the resultant conceptual schema

The **resultant conceptual schema is correct according to the expectations formalized** in the processed test cases (the knowledge included in the conceptual schema fulfills the expectations formalized as test case assertions).

The resultant schema is also complete according to the test set, because the knowledge it contains makes possible the test set execution.

However, more user stories could be designed and, consequently, more test cases could be specified in order to increase our confidence about the correctness and the completeness by testing the schema in more representative cases. This is a drawback inherent to all the testing processes, because the number of possible test cases is infinite. In this case study, we learned that it is very important to specify the test cases based on a representative set of user stories according to a planned testing strategy.

All knowledge defined in the **resultant conceptual schema is relevant**. The passing test set and its associated conceptual schema are not enough to assert the relevance of the schema (the defined knowledge is correct and necessary, but the schema could contain irrelevant knowledge that does not alter the verdict of the test cases). However, the CSTL processor allows to automatically check whether the basic types, derived types, valid type configurations or domain event types are participants of any test case or not. If all the elements of the schema are needed in at least one correct test case, it is because the defined knowledge is relevant for the system.

Figure 4 shows the coverage analysis report provided by the CSTL Processor at the end of the last iteration. It allows us to ensure the relevance of the knowledge defined in the resultant conceptual schema.

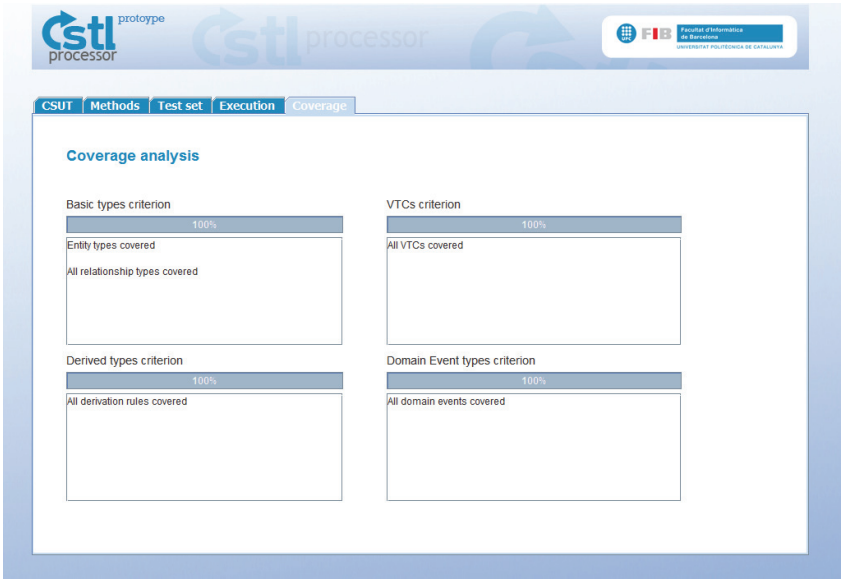


Fig. 4. Coverage report provided by the CSTL processor at the end of the last TDCM iteration

4.2. Errors and failures

Errors/failures categorization

In this case study we have categorized the errors and failures which may be obtained during the execution of test cases by applying TDCM. Neither syntactical errors nor incorrectly formalized expectations in test cases are considered in this table. Figure 5 summarizes the categorization of errors/failures which has been used and refined. We also suggest the applicable changes to fix each error/failure type. This categorization assumes that all the states during the execution of a test case are achieved by asserting the occurrence of domain events. This is a common assumption when applying TDCM in order to develop a conceptual schema which defines both the structural and the behavioral knowledge.

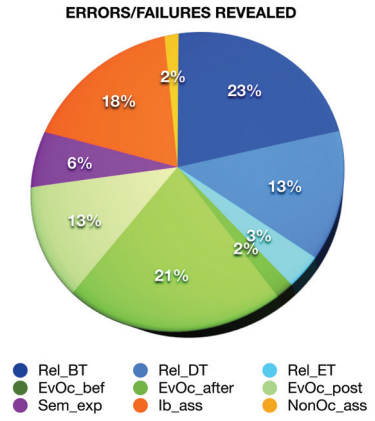
Code	Error/Failure	Description	Fixing actions	Fixing action code
Rel_BT	ERROR	A basic type involved in a test case does not exist in the CSUD	The basic type is relevant and it is added to the CSUD.	*Add_Rel_BT*
Rel_DT	ERROR	A derived type involved in a test case does not exist in the CSUD	The derived type is relevant and it is added to the CSUD.	*Add_Rel_DT*
Rel_ET	ERROR	A domain event type involved in a test case does not exist in the CSUD	The domain event type is relevant and it is added to the CSUD.	*Add_Rel_ET*
EvOc_bef	ERROR	Inconsistent state before the occurrence of an event	Some static constraint is invalid and it is modified.	*Chg_constraint*
			Some event initial integrity constraint (precondition) is invalid and it is modified	*Chg_pre*
EvOc_after	ERROR	Inconsistent state after the occurrence of an event	The event postcondition or method is incorrect and it is modified.	*Chg_post_method*
			Some constraint is invalid and it is modified.	*Chg_constraint*
EvOc_post	ERROR	The postcondition of an event is not satisfied.	The method is not correct and it is modified.	*Chg_method*
			The postcondition is not correct and it is modified.	*Chg_post*
Sem_exp	ERROR	Semantic error in an expression (e.g. an operation that requires an order set is applied to a unordered set, an addition is applied to a non-numeric element, etc.).	The expression is corrected.	*Chg_exp*
			The semantic error reveals that the CSUD needs not yet specified knowledge. The type of an element of the CSUD is change to make the expression feasible.	*Chg_element_type*

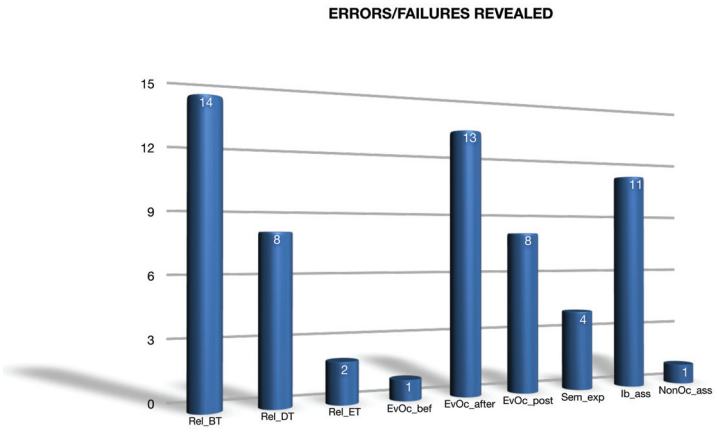
IB_ass	FAILURE	An assertion about the Information Base (IB) state fails.	The effect of an event type is not correct and it is changed.	*Chg_post_method*
			A derivation rule is incorrect and it is changed.	*Chg_der_rule*
NonOc_ass	FAILURE	The non-occurrence assertion of an event fails.	An event initial integrity constraint (precondition) is added/updated.	*Chg_pre*

This categorization and their associated actions may be useful guidelines to help making progress in TDCM more efficiently. In TDCM, errors and failures to be fixed are the essence for progress. When errors/failures are revealed, then the modeler may use this table to find out applicable actions to change the schema in order to fix each error/failure.

In the case study, the errors/failures which drive the changes in each iteration are reported and classified using this categorization.

Errors and failures that drive conceptual modeling in the case study





In the previous charts we analyze the errors and failures revealed by applying TDCM to the bowling game case study, according to the errors/failure categorization described in the previous section.

We observe that TDCM drives the development of the bowling game conceptual schema by promoting to fix three main kinds of errors/failures:

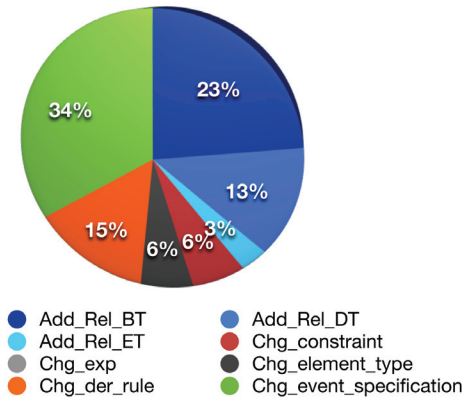
- 39% of the errors/failures correspond to relevant types (basic types, derived types or event types) which have not been defined yet in the schema (Rel_BT+Rel_DT+Rel_ET). Rel_BT, Rel_DDT and Rel_ET are proportional to the relevant knowledge to be defined in the schema. Note that the bowling game case study has two domain events, but the number of domain events is usually greater when developing the conceptual schema of an information system.

- 36% of the errors/failures correspond to erroneous definitions of domain event types (EvOc_bef+EvOc_afterçEvOc_post), either because the state before the occurrence is inconsistent, or because the state after the occurrence is inconsistent or because the postcondition is not satisfied.

- 20% of the errors correspond to unexpected results (assertions that fail). Most of them are assertions about the IB state (18%) and others correspond to assertions about the non-occurrence of events (2%). It is important to note that in this case study, there are only two domain events with a low number of restrictions about the application of domain events. It seems that in other domains, the assertions about the non-occurrence may be greater.

In the bowling game case study, some iterations have been driven by other semantic errors in OCL expressions (basically, operations which are applied to invalid element types).

CHANGES TO THE CONCEPTUAL SCHEMA BY APPLYING TDCM



The errors and failures which are reported by the CSTLProcessor in each iteration need to be fixed according to the TDCM cycle. Changing the schema to fix the errors/failures makes progress in the incremental development of the schema. By analyzing the kinds of actions applied to fix the previously analyzed errors/failures in the bowling game system, we observe:

- Fixing the errors about missing relevant knowledge are almost trivial: they need to be added. Note that the percentage of the changes Add-Rel_BT (23%), Add_Rel_DT (13%) and Add_Rel_ET (3%) are exactly the same as the errors revealed due to missing relevant knowledge in the conceptual schema.

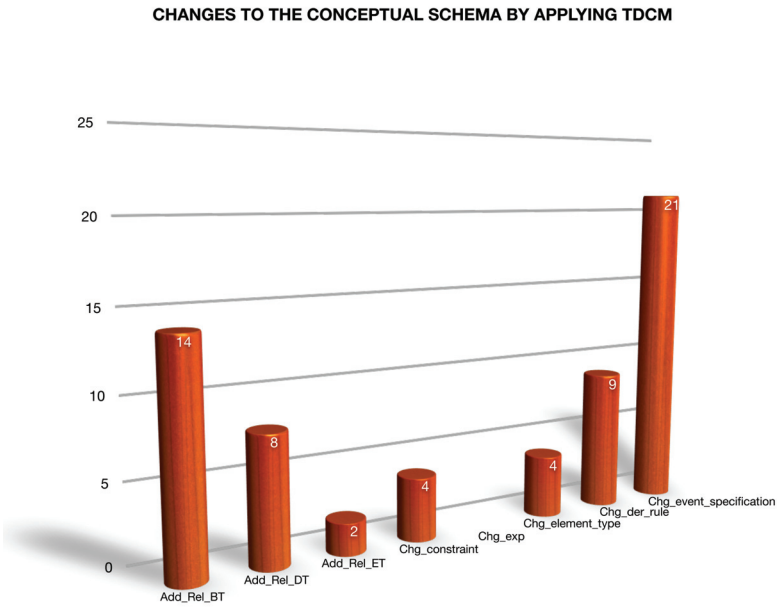
- We observe that most of the changes (34%) correspond to the refinement of event specifications (precondition, postcondition, method) in order to be correctly defined according to the general definition of domain events.

- Derivation rules have also an important role. 15% of the changes in the schema by applying TDCM in the bowling game system correspond to changes in derivation

rules. These changes may be induced by failing assertions about the IB state, by failing postconditions or methods, etc.

- 6% of the changes correspond to the addition/refinement of static constraints of the schema. These changes are usually induced by invalid IB states when an event occurs.

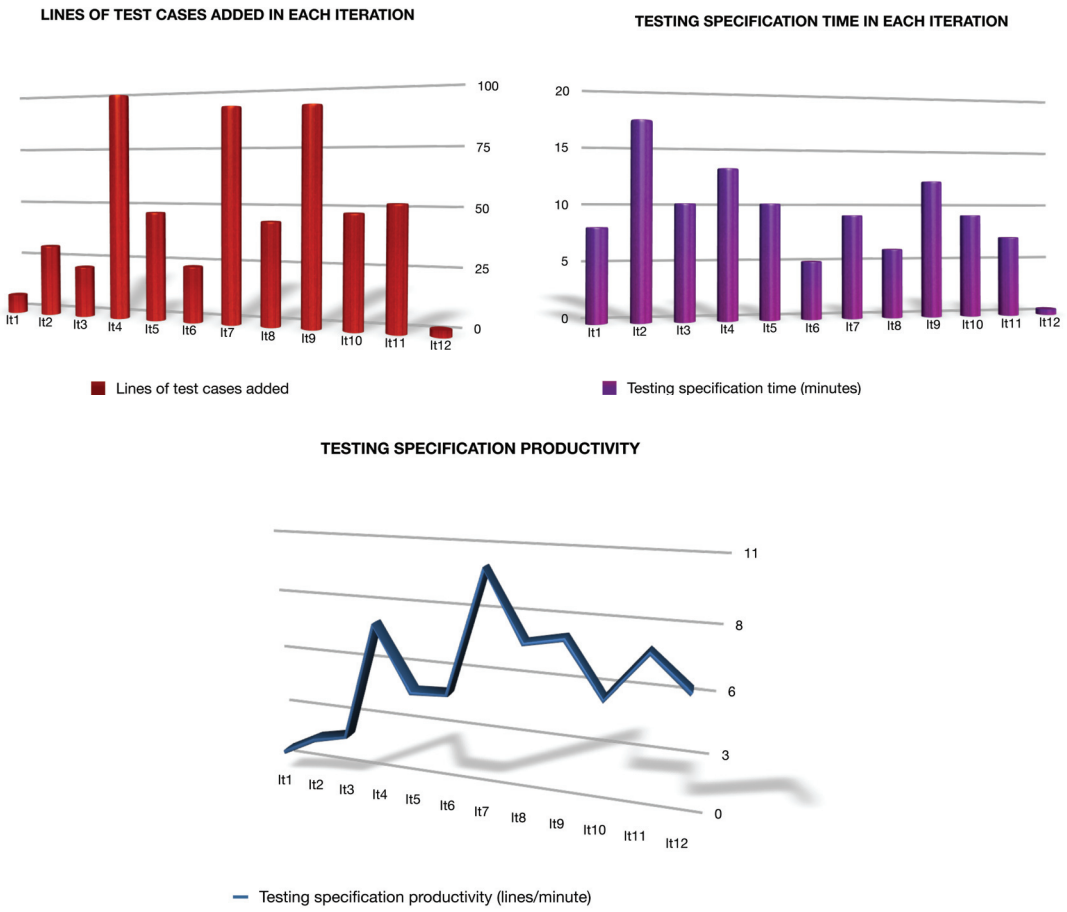
- Semantic errors in expressions reveal inconsistencies in the schema. They need to be corrected either by changing the expression or changing the type of an element of the schema in order to make possible the evaluation of the expression. In the bowling game case study, all of these errors have been corrected by changing the type of a schema element.



It is important to note that this results point out a tendency about the most common errors/failures and their induced changes by analyzing the application of TDCM in a concrete case study. However the kind of errors/failures revealed and the changes driven by TDCM also depend on the knowledge of the universe of discourse of the system for which we develop the conceptual schema.

4.3. Iterations analysis

In this section, we analyze and compare the 12 iterations (we name them as it1, it2, it3, it4, it5, it6, it7, it8, it9, it10, it11, it12) that have been performed by applying TDCM to the bowling game case study. The complete report about each specific iteration is described in Section 3.

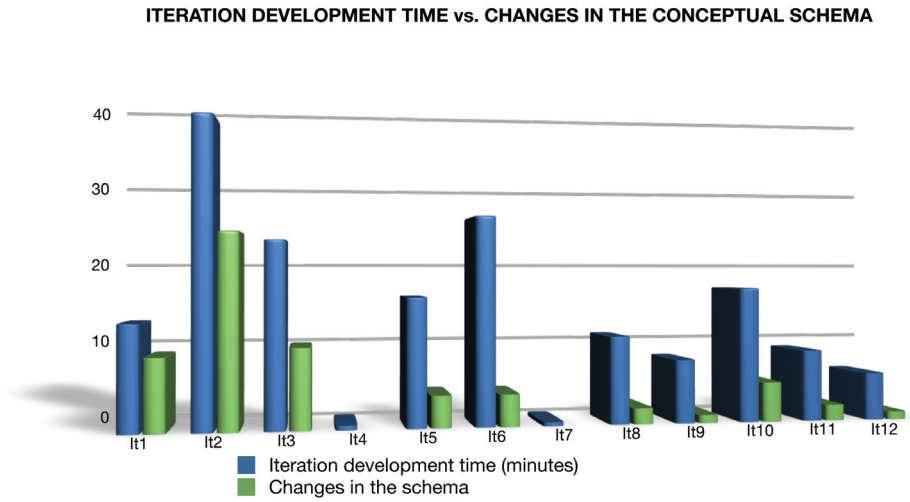


Lines of test programs in each iteration.

As we can observe in the first bar chart above, the numbers of lines of test cases added to the test set vary in each iteration. Larger test cases correspond to stories that represent sequences of throws in a complete bowling game instance. In the other iterations, incomplete games are formalized in order to test specific situations according to the user stories. Note that *It1* is the first fragment of a test case, which is extended in *It2*. Similarly, *It12* ads only three lines of testing code to extend again the first test case.

If we compare the first bar chart (lines of test cases added) with the time spent by specifying the test cases in each iteration (the second bar chart), we can observe that the testing specification time is not directly proportional to the lines of test cases added in all cases.

The third chart above represents the testing specification productivity (lines of test added/minute). We observe that, in general, the productivity tends to increase as we make progress in the TDCM application. We also realize that there are peaks of productivity in the iterations that reuse previously used testing structures. In contrast, the testing specification consumes more time when we specify stories with new (and probably unknown) structures.



In the previous bar chart we compare the development time used to complete each TDCM iteration with the changes applied to the schema due to the fixing actions induced by the revealed errors/failures.

We can observe that, in most of the iterations, the time spent by fixing errors/failures is proportional to the number of changes in the conceptual schema. However, it is important to observe that there are iterations in which the time spent is not as productive (in terms of performed changes per minute) as in other iterations. This analysis suggests that not all the errors require the same effort to be analyzed and fixed. We will analyze it better with the following charts.

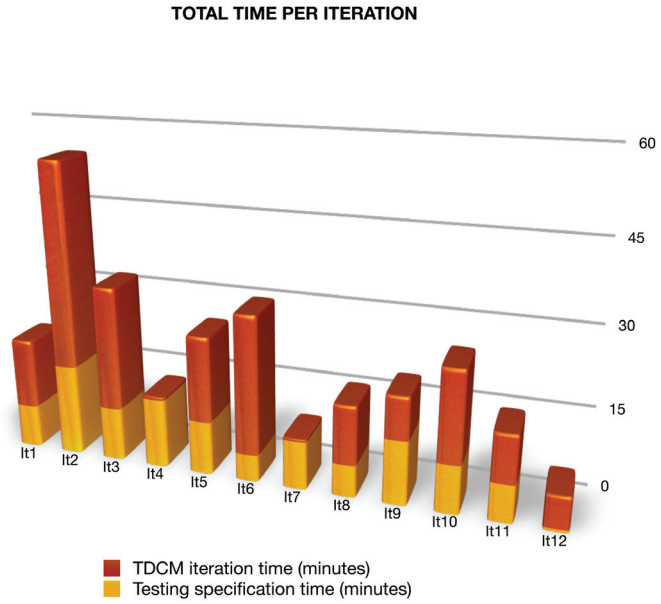
It is important to note that there are two iterations (*It4* and *It7*) with insignificant iteration time spent and without changes in the schema. It means that the verdict of the test set is *Pass* from the first execution and, consequently, the iteration does not make progress in the TDCM cycle. Nevertheless, these iterations increase our confidence about the validity of the schema.

The time spent in a TDCM iteration is the time to fix the errors/failures (that is to evolve the schema), but we should note that at the end of each iteration, we obtain an executable conceptual schema with a test set that validates its correctness and completeness.

It is also remarkable the use of previous test cases for regression testing purposes. In iteration 5 there is an example. A previous test case fails but we efficiently discover the reason because we can analyze the line that fails. In general, in the bowling game case study, there is no so much reconsideration of previous knowledge added, but when reconsideration is needed to fix the current error (as in iteration 5), then the time spent is not significantly increased in comparison with fixing an error in the current test case. Going back to question previous knowledge is very common in conceptual modeling. In TDCM, regression testing helps minimizing them. Defining a testing strategy that pursues an order of processing of the stories based on its expected complexity, may contribute to

reduce going backs in the development as shown in this case study (there is only one going back case).

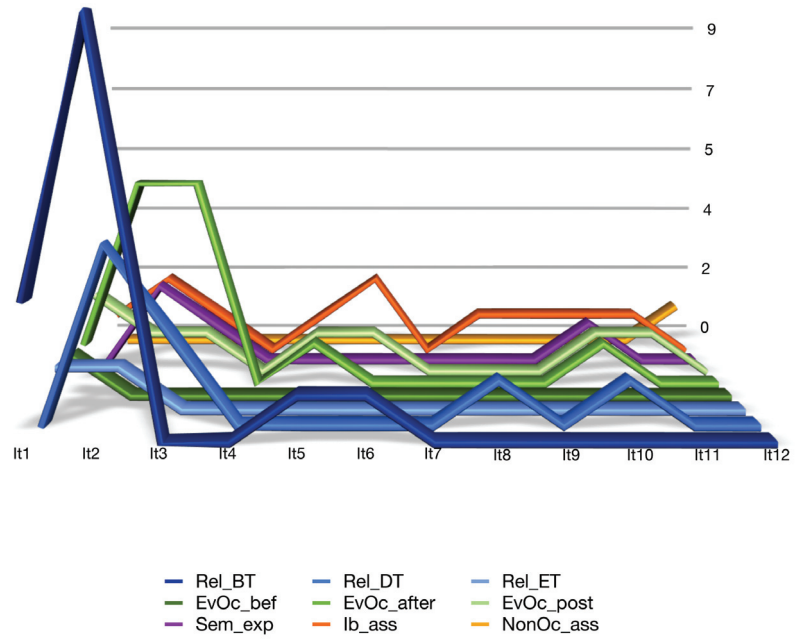
We have also automatically analyzed, at the end of each iteration, the basic types, the derived types and the domain event types which have been tested in at least one of the passing test cases of the current test set. This analysis gives as a measure of a basic coverage of the test set and allows us to identify elements in the schema that has not been tested. When applying TDCM in the case study of the bowling game system, we realize that, at the end of each iteration, the coverage is 100% (all the elements of the schema are participants of a valid and passing test case). The coverage report shown in Fig. 4 it has been obtained once each iteration is finished. Therefore, for all the elements of the schema, at the end of each iteration, its relevance is justified by the test set.



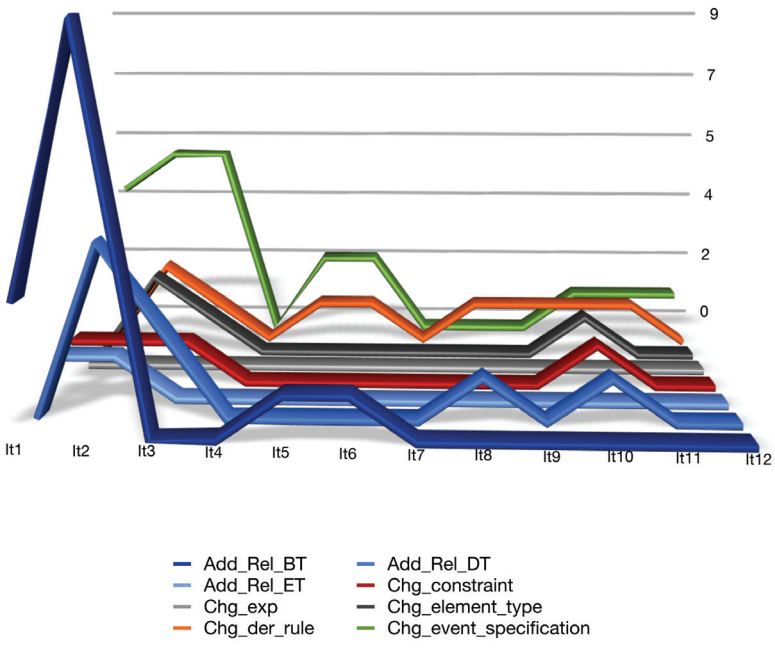
The previous accumulated bar chart represents the total time spent in each iteration. In most iterations, the time spent in the TDCM development (fixing errors and failures) is greater than the time spent in fixing errors/failures and changing the schema. It means that in most of the iterations, the testing specification time worth the while because the test case encourages and drives the evolution of the conceptual schema.

Again, the exceptions are the iterations *It4* and *It7*. In these iterations, the TDCM iteration time is insignificant, because no changes are done in the schema (these are iterations that pass in the first execution). Therefore, it seems that the productivity (in the context of TDCM) of the iterations *It4* and *It5* is very low (they increase the confidence on the validity of the system but they not drive the evolution of the schema). In these iterations, the time spent by designing and specifying the test case is higher in comparison with the TDCM iteration time spent.

ERRORS/FAILURES REVEALED IN EACH ITERATION



CHANGES APPLIED IN EACH ITERATION



Finally, the two previous charts, allows us to analyze the distribution of the different kinds of errors/failures and the changes which have been performed to fix them while TDCM is applied.

We can observe that in the first iterations the main errors found correspond to relevant types that are not in the schema. It is required to add them to the conceptual schema. As we add the first domain events, we detect inconsistent states that require refining static constraints and correctly specifying the effect of these events. After the peak of the first initial knowledge to be defined, the following iterations reveal more specific errors (incorrect derivation rules, inconsistent expressions, etc.) that may also lead to refine some of the events or adding some particular knowledge.

The previous charts also support the conclusion that not all kinds of knowledge require the same effort to be evolved or corrected according to the processed test cases. In the first iterations, the number of errors/failures is greater because we basically add relevant knowledge. After that, when we specify the effect of the events and we make assertions about derivation rules or the IB states reached by the events, the required effort is greater because it is less evident how to change the schema in order to reach the verdict *Pass*.

5. Conclusions

- **We have applied Test-Driven Conceptual Modeling to a popular case study** (the bowling game system) used to illustrate eXtrem Programming practices in some books and resources.

- After the application of TDCM in this case study, we obtained **an executable conceptual schema with a test set that checks the correctness and the completeness of the schema** according to the expectations formalized as tests. In other words, TDCM iterations drive the evolution of the conceptual schema and continuously perform its validation.

- The time spent to specify the test cases in each iteration varies depending on the formalized stories. However, we observe that **testing structures are reused and, therefore, the testing specification productivity tends to increase** as we make progress in TDCM.

- The time spent in the conceptual schema evolution (by fixing errors/failures) is greater than the time used to specify the test cases. Therefore, **most of the test cases are productive because they lead to make progress in the evolution of the schema**. The exception are those iterations that pass in the first execution (they increase our confidence about the validity of the schema, but they do not drive changes).

- At the end of each TDCM iteration of the bowling game case study, the basic coverage of the elements of the schema is 100%. It means that, **at the end of each iteration, for all the elements of the schema, its relevance its justified and, in at least one case, its correctness has been tested**.

- **The most common errors/failures** revealed correspond to missing relevant types, to invalid definitions of domain event types and to failing assertions (either due to incorrect domain event effects, invalid static constraints or incorrect derivation rules).

- In first iterations, the most common errors are about missing relevant types (which are necessary to build IB states). After that, the most common errors/failures are about the correct definition of domain event types and the correctness (according to the assertions) of the reached IB states and the result of the derivation rules.

- The time spent on fixing errors/failures is not proportional in all cases to the number of errors fixed in each iteration. It means that **not all errors/failures require the same effort to be fixed**. The analysis suggests that missing relevant types are trivial to be fixed (they need to be added). However, the changes to fix failing assertions about the state of the domain or incorrect domain event specification may require different actions such as changing derivation rules, integrity constraints or the precondition and postcondition of the effect of the event.

- We have **identified and categorized the errors and failures that may be revealed and the associated changes in the schema that may be applied to fix them**. This categorization may help in the application of TDCM.

6. References

1. Martin, R. C. Agile software development: principles, patterns, and practices. Prentice Hall PTR Upper Saddle River, NJ, USA (2003)
2. Martin, R.C., Koss, R.S. An Extreme Programming episode. http://www.objectmentor.com/resources/articles/xpepisode.htm
3. Martin, R.C., Melnik, G., Inc, O.M. Tests and Requirements, Requirements and Tests: A Möbius Strip. IEEE Software 25(1), 54-59 (2008)
4. Meyer, B. Seven Principles of Software Testing. IEEE Computer 41(8), 99-101 (2008)
5. Tort, A. Testing the osCommerce Conceptual Schema by Using CSTL. Research Report LSI-09-28-R, UPC, http://hdl.handle.net/2117/6289 (2009)
6. Tort, A., Olivé, A. An approach to testing conceptual schemas. Data & Knowledge Engineering 69(6), 598-618 (2010)